Profitips und Techniken für Apple-UCSD-PASCAL



Robert Tolksdorf



Robert Tolksdorf

Profitips und Techniken

für

Apple-UCSD-Pascal



CIP-Kurztitelaufnahme der Deutschen Bibliothek

Tolksdorf, Robert:
Profitips und Techniken für Apple-UCSD-Pascal /
Rober Tolksdorf. - Gensingen : Luther, 1985
ISBN 3-620-00103-0

Titelseite: Peter Spann, Karlsruhe

Bei der Zusammenstellung wurde mit größter Sorgfalt vorgegangen. Fehler können trotzdem nicht vollständig ausgeschlossen werden, so daß weder der Verlag noch der Autor für fehlerhafte Angaben und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Warennamen sowie Marken- und Firmennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Für Verbesserungsvorschläge und Hinweise auf Fehler ist der Verlag dankbar.

6531 Gensingen, Printed in Germany
** ISBN 3-620-00103-0 **

Inhaltsverzeichnis

1. Einleitung

Dieses Buch wendet sich an den fortgeschrittenen Apple Pascal Programmierer. Es soll kein Lehrbuch sein, vielmehr sollen hier die Tips und Techniken beschrieben werden, die es ermöglichen, das System voll auszunutzen. Es bietet das notwendige Wissen, um auch intern in das System einzugreifen. Es gibt darüber wenig Literatur, da dies im Grunde genommen der Pascal-Philosophie widerspricht. Jedoch stößt man leicht an Grenzen, die es eigentlich garnicht geben müßte. Man will Dinge realisieren, die z.B. in Applesoft-Basic möglich und verbreitet sind. Hier schränkt das System den Programmierer ein. Hier setzt dieses Buch an, indem das Know-How vermittelt wird, daß eben nicht in den Handbüchern steht.

Das Buch ist in drei Abschnitte gegliedert. Im Ersten wird auf das Betriebssystem eingegangen. Es wird beschrieben, wie man direkten Zugriff auf die Diskette und auf das Directory erhält. Es wird eine Brücke geschlagen zwischen Apple-DOS 3.3 und dem Pascal-Betriebssystem. Schließlich wird auf das Format der Filetypen eingegangen und ein neuer hinzugefügt.

Der zweite Abschnitt geht näher auf die Programmierung in Pascal ein. Hier ist eine Zusammenstellung aller Systemroutinen und deren Adressen enthalten. Mit erweiterten "PEEK/POKE" Routinen erhält man Zugriff auf die internen Systemvariablen. Es wird beschrieben, wie man den Tastaturpuffer umgeht, und wie man Zeichen in diesen ohne Tastatureingabe setzt. Schließlich wird dargestellt, wie man auch unter Pascal das Drücken der Reset-Taste abfangen kann.

Der dritte Abschnitt geht auf Grafik und Text ein. Es werden Programme vorgestellt zum Editieren von Grafiken und zum Verändern des Graphikzeichensatzes. Schließlich wird die unter Basic bekannte Blockgrafik auch in Pascal ermöglicht, wobei auch Shapes realisiert werden. Es wird ein Mangel des Text-Editors beseitigt, und es wird ein Programm zum schnellen Messen der Textgröße verwirklicht.

In den Kapiteln wird zunächst geschildert, welches Problem gelöst werden soll, und wie der Weg dorthin aussieht. Daran schließt sich ein fertiges Programmlisting an. Im Zusammenhang mit den Erläuterungen sollte es dem Programmierer möglich sein, die Programme speziell zu verändern, oder in seinen Projekten weiterzuverwenden.

Es soll an dieser Stelle darauf hingewiesen werden, daß das Eingreifen in das System nicht der Pascal-Philosophie entspricht. Sie fordert, daß ein Programm, daß auf dem Rechner A unter Pascal läuft, auf den Rechner B ohne Anpassungen übertragen werden kann. Verwendet man die hier geschilderten Tricks, so wird dies nicht immer der Fall sein.

Wie der Leser mit diesem Buch umgeht, bleibt ihm überlassen. Es ist nicht nötig, das Buch von vorne nach hinten durchzuarbeiten. Es ist jedoch zu empfehlen, zunächst den erläuternden Text in jedem Kapitel zu lesen und dann erst das Programm einzutippen.

Dabei viel Spaß !

2. Utilities für das Betriebssystem

In den folgenden Kapiteln sollen einige Utilities vorgestellt werden, die die Struktur und die Feinheiten des Pascal Operating Systems (im folgenden "POS" genannt) ausnützen, und damit neue Möglichkeiten bieten. Es geht hier ausschließlich um die Arbeit mit den Disk-Laufwerken, d.h. um die Behandlung von Files und Daten auf Disketten. Voraussetzung ist, daß dem Leser die Arbeitsweise von POS bekannt ist, und er mit den Dienstprogrammen, wie Editor oder Filer umgehen kann.

2.1 Block-Editor

Disketten sind gewöhnlich in sogenannte Blöcke und Sektoren eingeteilt. Sie stellen die kleinste Dateneinheit dar, die von der Floppy gelesen werden kann.

Mit Sektoren bezeichenen wir die Datenmenge, die auf einmal von der Diskette gelesen wird. Sie ist größtenteils von dem verwendeten Laufwerk abhängig. Ein logischer Block ist die Datenmenge, mit der das Betriebssystem arbeitet. Diese beiden Einheiten sind von einander unabhängig und müssen nicht die gleiche Größe haben. Es ist Aufgabe des Betriebssystems, dafür zu sorgen, daß die richtige Anzahl von Sektoren für einen logischen Block gelesen wird.

Mit logischen Blöcken ist die Datenmenge gemeint, die vom Betriebssystem auf einmal gelesen wird. Die Größe dieser Blöcke hängt vom verwendeten Betriebssystem ab. So gibt es Blöcke, die 128 Byte enthalten (z.B. CP/M), das Apple-DOS 3.3 verwendet Blöcke mit 256 Bytes und POS schließlich kennt 512 Bytes große Diskettenabschnitte. Es gibt unter anderen Betriebssystemen auch größere Einheiten, wie 1024 Bytes.

In unserem Fall, der Arbeit mit einer Apple-Maschine beträgt die Sektorengröße des Laufwerks 256 Bytes und die der logischen Blöcke von POS 512 Bytes. Uns interessiert im folgenden nur die Verwendung von Blöcken. Das Umrechnen in Sektoren überlassen wir dem Betriebssystem.

Mit einem Block-Editor wollen wir es uns ermöglichen, auf jedes Byte eines Blocks einer Pascal-Diskette zuzugreifen, es zu verändern und den Block wieder zurückzuschreiben. Wir können damit die Struktur der Diskettenorganisation erforschen, oder wir können Veränderungen in Programmen vornehmen. Bei letzterem liegt unser Augenmerk darauf, daß z.B. in einem größeren Programm eine falsch geschriebene Bildschirmmeldung ohne Neu-Compilierung berichtigt werden kann. Änderungen in den Programmen selber sollten nicht durchgeführt werden.

Unsere allgemeine Zielsetzung steht nun fest, wir sammeln nun, was wir von unserer Lösung erwarten.

Wir wollen eine Ausgabe des Block in hexadezimaler Schreibweise und in ASCII, d.h in alphanumerischen Zeichen (ASCII = American Standard Code for Information Interchange) beinhaltet. Dann wollen wir natürlich einen beliebigen Block lesen und schreiben können. Weiterhin sollte der ganze Block mit einem bestimmten Wert zu füllen und schließlich jedes einzelne Byte zu verändern sein.

Der Editor soll einen guten Überblick über einen Block bieten. Also muß das Schirmbild so gut wie möglich sein. Wir haben 512 Bytes in hexadezimaler Schreibweise auszugeben. Da diese zusammen mit einer ASCII-Ausgabe nicht genug Platz auf dem Bildschirm haben, müssen wir die Ausgabe teilen. Es werden immer 256 Bytes gleichzeitig dargestellt. Der Benutzer hat dann über ein Kommando die Möglichkeit, zwischen der ersten und der zweiten Hälfte hin und her zu schalten.

Die 256 Bytes werden in 16 Zeilen zu je 16 Bytes darstellt. Da auch eine ASCII-Ausgabe verlangt wird, hängen wir einen 16 Zeichen langen String an. Am Anfang der Zeile steht eine Adresse, die die Position der Bytes in dem 512 Bytes großen Feld angibt. Damit ergibt sich folgendes Zeilenlayout:

```
000:  00 01 02 03 04 05 06 .. 0C 0D 0E 0F !"#$%&'()0*=!"#$

Pos.            Bytes Pos+0..15             16 Zeichen in ASCII
```

Wir haben also schon 16 Zeilen belegt, es bleiben uns noch 8 Zeilen übrig, die wir frei verwenden können. Eine Zeile nehmen wir als Menüzeile, eine als Informationszeile und eine als Eingabezeile.

Um die Information lesen und schreiben zu können benutzen wir die Standard-Prozeduren "UNITREAD" und "UNITWRITE". Im Pascal-Handbuch ist die Bedeutung der Parameter erläutert. Wir können eine vom Benutzer angegebene Blocknummer einsetzen und den Inhalt des Blocks in ein Feld einlesen.

Das Füllen des Blocks mit einem bestimmten Wert ist einfach. Der Benutzer gibt eine Zahl ein, das ARRAY wird mit diesem Wert aufgefüllt, und der Benutzer braucht nur noch den Block zurückzuschreiben.

Zum Verändern einzelner Bytes ist es nötig, daß der Benutzer angibt, welches Byte er ändern will, und einen Wert, der diesem Byte zugewiesen werden soll. Daraufhin wird das Feld geändert und auf dem Schirm die hexadezimale und die ASCII-Ausgabe erneuert.

Schließlich werden noch zwei zusätzliche Kommandos eingebaut. Als erstes ein Kommando zum Verlassen des Programms und dann eine Hilfs-Funktion, die eine Kommandoliste ausgibt.

Wichtige Variablen

"BLOCK" : 512 Byte großes Feld zur Aufnahme eines Blocks

Wichtige Prozeduren

"HOLECHAR" : Liest ein Zeichen von der Tastatur ein, das aus "OKSET" sein muß. So werden schon bei der Eingabe nicht vorhandene Kommandos ausgeschlossen.

"HOLESTRING" : Liest einen String von der Tastatur ein. Die Zeichen müssen aus "ALLOWED" sein und die Eingabe hat die maximale Länge "LEN".

"BYTTOHEX" : Wandelt ein Byte in einen String

hexadezimaler Schreibweise um.

"HEXTOINT" : Wandelt einen String, der eine hexadezimale Zahl enthält in einen Integerwert um.

"LESEBLOCK" : Liest mit "UNITREAD" einen Block von der Diskette in "BLOCK" ein.

"SCHREIBBLOCK" : Schreibt "BLOCK" mit "UNITWRITE" in einen Block auf der Diskette.

Kommandoliste

"O" : obere 256 Bytes des Blocks anzeigen
"U" : untere 256 Bytes des Blocks anzeigen
"L" : einen Block von der Diskette lesen
"S" : den Block auf die Diskette schreiben
"N" : den gleichen Block nochmals lesen
"A" : den Block an die gleiche Stelle auf der Diskette zurückschreiben
"+" : nächsten Block lesen
"-" : Block davor lesen
"V" : ein Byte verändern
"F" : Block mit einem Wert füllen
"?",
"/" : Kommandoliste ausgeben
"E" : Blockeditor verlassen

Programm "BEDIT.TEXT"

```
programm blockedit;
type
  setofchar = set of char;

var
  home,cr,bs,eol,bell : char;
  hexdigit : packed array[0..15] of char;
  block:packed array[0..511] of 0..255;
  blkno:integer;
  c:char;
  cmdset:setofchar;
  oben:boolean;

  procedure init;  { Initialisiert Variablen und bringt den }
  var i:integer;   { Darstellungsrahmen auf den Bildschirm  }
  begin
    eol:=chr(29);
    bell:=chr(7);
    cr:=chr(13);
    bs:=chr(8);
    home:=chr(12);
    hexdigit:='0123456789ABCDEF';
    cmdset:=['O','o','U','u','L','l','S','s','N','n','A','a',
             '+',';','=','-','V','v','F','f','E','e','?','/'];
    write(home);
    gotoxy(0,2);
    write(' Adr 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F');
    gotoxy(0,4);
    for i:= 0 to 15 do
```

```
    writeln('   ',hexdigit[i],'0');
  blkno:=0;
end;

function hole_char(okset:setofchar):char; { liest ein Zeichen ein, das  }
var                                      { in okset enthalten sein mu~ }
  ch   : char;
  gut  : boolean;
begin
  repeat
    read(keyboard,ch);
    if eoln(keyboard) then ch:=cr;
    gut := ch in okset;
    if not gut then write(bell)
      else if ch in [' '..chr(125)] then write(ch);
  until gut;
  hole_char:=ch;
end;

procedure warte_auf_ret; { Wartet auf die Eingabe eines returns }
var c:char;
begin
  gotoxy(0,23);
  write('Bitte <ret> druecken');
  c:=hole_char([cr]);
end;

procedure fehler; { Gibt Fehlermeldung aus }
var i:integer;
begin
  gotoxy(0,21);
  write(bell,'*** I/O Fehler ***');
  for i:= 1 to 3000 do;
  gotoxy(0,21);
  write(eol);
end;

procedure hole_string(var s:string; allowed:setofchar; len:integer);
var c:char;      { Liest des String s ein. Seine maximale Laenge steht in len }
hilfss:string;   { Alle Zeichen muessen aus allowed sein.                     }
begin
  s:='';
  hilfss:=' ';
  repeat
    c:=hole_char(allowed+[bs,cr]);
    if c=bs then if length(s)=0 then write(bell)
                              else
                                begin
                                  s:=copy(s,1,length(s)-1);
                                  write(bs,' ',bs);
                                end;
    if c in allowed then if length(s)=len then write(bell,bs,' ',bs)
                                         else begin
                                                hilfss[1]:=c;
                                                s:=concat(s,hilfss);
                                              end;
    if c=cr then if length(s)=0 then
                                begin
                                  write(bell);
                                  c:=' ';
                                end;
```

```
  until c=cr;
end;

function hole_blkno(prompt:string):integer; { liest eine Blocknummer ein }
var i,num:integer;
      s:string;
     ok:boolean;
begin
  write(prompt,eol);
  repeat
    hole_string(s,['0'..'9'],3);
    num:=0;
    for i:=1 to length(s) do num:=num*10+(ord(s[i])-48);
    if num<280 then ok:=true
               else begin
                      for i:=1 to length(s) do write(bs);
                      for i:=1 to length(s) do write(' ');
                      for i:=1 to length(s) do write(bs);
                      ok:=false;
                    end;
  until ok;
  hole_blkno:=num
end;

procedure byttohex(byt:integer; var hexstr:string); { Wandelt den Wert byt }
begin                     { in den String hexstr in hexadezimaler Notierung um }
  hexstr:=' 00';
  hexstr[2]:=hexdigit[byt div 16];
  hexstr[3]:=hexdigit[byt mod 16];
end;

function hextoint(hexstr:string):integer; { wandelt den hexadezimalen String }
var                                        { hexstr in einen Integerwert um }
  i,num,digit:integer;
begin
  num:=0;
  for i:=1 to length(hexstr) do
    begin
      digit:=scan(16,=hexstr[i],hexdigit);
      num:=num*16+digit;
    end;
  hextoint:=num;
end;

function hole_hexval(prompt:string;maxlen:integer):integer; { liest einen }
var                                                 { hexadezimalen Wert ein }
  s:string;
begin
  write(prompt,eol);
  if maxlen>4 then maxlen:=4;
  hole_string(s,['0'..'9','A'..'F'],maxlen);
  hole_hexval:=hextoint(s);
end;

procedure gebe_aus(start:integer); { gibt den Inhalt des Blocks aus }
var dmp,str:string;
    inh,j,i:integer;
begin
  if start=0
    then inh:= 32
    else inh:= 49;
```

```
  for i := 4 to 19 do
    begin
      gotoxy(1,i);
      write(chr(inh));
    end;
  oben:=(start=0);
  dmp:=' 0123456789123456';
  for i:= 0 to 15 do
    begin
      gotoxy(4,4+i);
      for j:= 0 to 15 do
        begin
          inh:=block[start+i*16+j];
          byttohex(inh,str);
          write(str);
          if inh>127 then inh:=inh-128;
          if (31<inh) and (inh<127)
            then dmp[j+2]:=chr(inh)
            else dmp[j+2]:='.';
        end;
      writeln(dmp);
    end;
  gotoxy(0,23);
  write('Blocknummer: ',blkno:4);
end;

procedure lese_block; { liest einen Block ein, und gibt ihn aus }
begin
  (*$I-*)
  unitread(4,block,512,blkno,0);
  (*$I+*)
  if ioresult<>0 then fehler;
  gebe_aus(0);
end;

procedure lese; { liest einen gewuenschten Block ein }
begin
  gotoxy(0,21);
  blkno:=hole_blkno('Welchen Block lesen ? ');
  gotoxy(0,21);
  write(eol);
  lese_block;
end;

procedure schreib_block; { schreibt einen Block auf Diskette }
begin
  (*$I-*)
  unitwrite(4,block,512,blkno,0);
  (*$I+*)
  if ioresult<>0 then fehler;
end;

procedure schreibe; { schreibt auf gewuenschten Block }
begin
  gotoxy(0,21);
  blkno:=hole_blkno('Welchen Block schreiben ? ');
  gotoxy(0,21);
  write(eol);
  schreib_block;
end;
```

```
procedure fuelle; { fuellt Block mit einem Wert }
var neu:integer;
begin
  gotoxy(0,21);
  neu:=hole_hexval('Womit fuellen ?',2);
  fillchar(block,sizeof(block),chr(neu));
  gotoxy(0,21);
  write(eol);
  gebe_aus(0);
end;

procedure vor; { liest naechsten Block ein }
begin
  blkno:=blkno+1;
  if blkno>279 then blkno:=279;
  lese_block;
end;

procedure zurueck; { liest einen Block vorher ein }
begin
  blkno:=blkno-1;
  if blkno<0 then blkno:=0;
  lese_block;
end;

procedure veraendere; { veraendert ein bestimmtes Byte }
var adr,new:integer;  { und gibt Veraenderung aus      }
    str:string;
begin
  repeat
    gotoxy(0,21);
    adr:=hole_hexval('Welches Byte ? ',3);
  until adr<512;
  gotoxy(0,21);
  write(eol,hexdigit[(adr div 256) mod 16],
            hexdigit[(adr mod 256) div 16],
            hexdigit[(adr mod 256) mod 16],': ');
  byttohex(block[adr],str);
  write(str,' ');
  new:=hole_hexval('',2);
  block[adr]:=new;
  gotoxy(0,21);
  write(eol);
  if (oben and (adr<256)) or
     ((not oben) and (adr>255)) then
    begin
      if adr>255 then adr:=adr-256;
      gotoxy(4+(adr mod 16)*3,4+(adr div 16));
      byttohex(new,str);
      write(str);
      gotoxy(53+(adr mod 16),4+(adr div 16));
      if new>127 then new:=new-128;
      if (32>new) or (new>126) then new:=46;
      write(chr(new));
    end;
end;

procedure hilf; { gibt Kommandoliste aus }
begin
  gotoxy(0,20);
  writeln('O)ben     U)nten    L)ese    S)chreibe');
```

```
      writeln('N)ochmals A)ktualisieren     V)eraendern');
      writeln('+) vor    -)zurueck F)uellen E)nde');
      warte_auf_ret;
      gotoxy(0,20);
      write(eol,cr,eol,cr,eol,cr,eol,'Blocknummer: ',blkno:4);
    end;

  begin
    init;
    lese_block;
    repeat           { Hauptschleife: liest Kommandos ein und fuehrt sie aus, }
      gotoxy(0,0); { bis E)nde gewaehlt wird                                  }
      write('Bedit: 0, U, L, S, N, A, +, -, V, F, E ',bs);
      c:=hole_char(cmdset);
      case c of
        'O','o': if (not oben) then gebe_aus(0);
        'U','u': if oben then gebe_aus(256);
        'L','l': lese;
        'S','s': schreibe;
        'N','n': lese_block;
        'A','a': schreib_block;
        '+',';': vor;
        '=','-': zurueck;
        'V','v': veraendere;
        'F','f': fuelle;
        '?','/': hilf
      end;
    until c in ['E','e'];
  end.
```

2.2 Das Directory

Um ein File von der Diskette lesen zu können, ist es gezwungenermaßen nötig, zu wissen, wo es steht, welchen Namen es hat sowie andere Zusatzinformationen. Unter POS sind sie in dem sogenannten Directory abgelegt. Dies ist ein Bereich auf der Diskette, der eine bestimmte Struktur hat. Man kann sich das Directory vom Filer aus mit "L" oder mit "E" auf dem Bildschirm ausgeben lassen. Ansonsten ist das Directory für Programme nicht zugänglich. Wenn wir jedoch wissen, wie das Directory aufgebaut ist und wo es auf jeder Diskette steht, wird es möglich, es von einem Programm aus einzulesen und zu verändern.

Das Directory steht in den ersten Blöcken einer jeden Diskette. Da der erste Block beim Booten benötigt wird, beginnt es bei Block zwei. Das Directory hat eine Größe von fünf Blöcken, so daß die ersten Files ab dem sechsten Block geschrieben werden.

Nun zum Aufbau des Directorys. Im Listing am Ende dieses Kapitels ist es als ein strukturierter Record dargestellt. Wir gehen nun diese Typendefinition Schritt für Schritt durch und erklären die Bedeutung der einzelnen Felder.

Jede Diskette hat einen Namen, über den sie angesprochen werden kann. Dieser Name kann bis zu sieben Zeichen lang sein. Wir definieren einen Diskettennamen als einen String mit der Länge 7 und nennen diesen Typen "VNAME", wie Volume-Name.

Jedes File hat einen Namen, der bis zu 15 Zeichen lang sein darf. Den entsprechenden Typen nennen wir "FNAME".

Weiterhin wird bei jeder Diskette und bei jedem Fileeintrag ein Datum geführt, das angibt, wann der jeweilige Eintrag zuletzt angesprochen wurde. Unter POS wird das Datum in einem gepackten Record dargestellt. Die Tage können von 0 bis 31, die Monate von 0 bis 12 und die Jahre von 0 bis 100 gehen. Dieser Record nimmt einen Platz von nur zwei Bytes ein.

Unter POS gibt es verschiedene Filetypen. Insgesamt sind es neun, wovon allerdings normalerweise nur fünf benutzt werden ("TEXT", "DATA","CODE","BAD","VOL"). Die Filetypen "INFO","GRAF","FOTO" und "SECR" werden in Apple-Pascal nicht unterstützt, sie sind aber im Directory vorgesehen. (In Kapitel 2.7 wird allerdings beschrieben, wie man den "FOTO"-Typ nutzen kann.) Wir fassen alle möglichen Typen als "FKIND" zusammen.

Nun kommen wir zu einem vollständigen Directory-Eintrag, "DIRENTRY". Zunächst wird bei jedem Eintrag vermerkt, wo sich das entsprechende File auf der Diskette befindet. Der erste Block wird in "FIRSTBLOCK", der letzte in "LASTBLOCK" angeben. Beidesmal handelt es sich um einen Integerwert.

Nun werden in einer "CASE"-Anweisung zwei Eintragstypen unterschieden. Sie sind abhängig von der Art des Files. Die erste Möglichkeit ist, das es sich um einen "VOL"- oder um einen "SECR"-Typ handelt. "VOL" bezeichnet einen Eintrag über das Directory und die Diskette selber. Der "SECR"-Typ wird auf dem Apple überhaupt nicht benutzt.

Handelt es sich also um einen "VOL"-Eintrag, so wird zunächst der Name der Diskette vermerkt. Es ist vom Typ "VNAME" und hat den Name "DISKVOL". Dann wird in einem Integerwert angegeben, wieviele Blöcke sich auf der Diskette befinden ("BLOCKNO"). Bei einem normalen Apple-Laufwerk sind dies 280, bei einem Fremdlaufwerk oft 320 oder mehr. Dann kommt die Angabe, wieviele Files sich auf der Diskette befinden ("FILENO"). Der nächste Eintrag "ACCESSTIME" wird auf dem Apple nicht genutzt und hat normalerweise den Wert 0. Er ist für uns uninteressant. Schließlich steht in "LASTBOOT", wann die Diskette zuletzt benutzt wurde. Bei jedem neuen Booten wird dieses Datum von Typ "DATEREC" eingelesen und als Systemdatum geführt. Es kann vom Filer aus geändert werden.

Bei einem normalen File-Eintrag kommt zunächst der Name des Files, "FILENAME" vom Typ "FNAME". Dann wird in "ENDBYTES" angegeben, wieviele Bytes im letzten Block genutzt sind. Schließlich ist in "LASTACCESS" das Datum angegeben, an dem das File zuletzt auf die Diskette geschrieben wurden. Damit ist der Eintrag komplett.

Das Directory eine Apple-Diskette hat 78 Einträge, wobei von 0 bis 77 gezählt wird. Der erste ist ein "VOL"-Eintrag, womit noch Raum für 77 Files bleibt. Wir fassen das ganze Directory in dem Typ "DIREC" als ein Feld mit 78 Einträgen zusammen.

Damit können wir das Directory von einem Programm aus ansprechen. Dies tun wir in dem abgedruckten Programm "DIR". Wir lesen das Directory ein, und geben es aus. Die Ausgabe entspricht der des Filer-Kommandos "E".

Mit "HOLEUNITNO" fragen wir den Benutzer, von welcher Unit wir ein Directory lesen sollen. Er kann dabei zwischen der Unit #4 und #5 auswählen.

"LESEDIR" liest das Direcory von der Diskette. Wir benutzen dazu die

"UNITREAD"-Anweisung. Es werden Daten von der angegebenen Unit in die Variable "DIRECTORY" gelesen, wobei bei Block Nummer 2 angefangen wird. Tritt ein Fehler beim Lesen auf, so wird das Programm mit einer Fehlermeldung abgebrochen.

In "GEBEDIRAUS" schließlich wird das Directory ausgegeben. Wir zeigen dabei alle Informationen an, die von Wichtigkeit sind. Zunächst den Namen der Diskette, das ihre Größe, die Anzahl der Files und das darauf verzeichnete Datum.

Dann wird bei allen vorhandenen Files der Name, die belegten Blökke, die Filegröße, das Datum, der Typ und schließlich die Anzahl der Bytes im letzten Block ausgeben. Dabei wurde darauf geachtet, daß der Bildschirm möglicht übersichtlich aussieht.

Man ist natürlich nicht darauf beschränkt, das Directory einzulesen. Wenn man es nach einer Änderung wieder zurückschreibt, kann man Funktionen des Filers von einem eigenen Programm aus durchführen. Dies wird im nächsten Kapitel demonstriert.

Wichtige Variablen

"DIRECTORY" : Variable von Typ "DIREC", die das gesamte Directory einer Diskette aufnimmt

"UNITNO" : Integerwert, der angibt, von welchem Laufwerk ein Directory eingelesen werden soll.

Wichtige Prozeduren

"HOLEUNITNO" : Fragt den Benutzer nach dem Wert für "UNITNO"

"LESEDIR" : Liest das Directory von der Diskette in die Variable "DIRECTORY" ein

"GEBEDIRAUS" : Gibt die Variable "DIRECTORY" strukturiert auf dem Bildschrim aus

Programm "DIR.TEXT"

```
Programm dir;
type
  vname=string[7];  { Ein Volumename }
  fname=string[15]; { Ein Filename }
  daterec = packed record { das Datum }
             month: 0..12;
             day:   0..31;
             year:  0..100;
            end;
  fkind = (vol,bad,code,text,info,data,graf,foto,secr); { moegliche Filetypen }
  direntry = record { ein Eintrag im Directory }
              firstblock: integer; { Block, bei dem das File anfaengt }
              lastblock: integer;  { Block, bei dem das File endet }
              case filekind: fkind of
                   { nur beim Eintrag #0 }
                   vol,secr: (diskvol: vname;   { Name der Diskette }
                              blockno: integer; { Anzahl der Bloecke }
                              fileno:integer;   { Anzahl der Files }
                              accesstime: integer; { unbenutzt }
                              lastboot: daterec); { Datum, an dem die      }
```

```
                                                    { zuletzt benutzt wurde }
                   { Normale File-Eintraege }
                   bad,code,text,
                   info,data,graf,
                   foto: (filename: fname; { Filename }
                          endbytes: 1..512; { Bytes im letzten Block }
                          lastaccess: daterec); { Datu, an dem das File }
              end;                                  { geschrieben wurde }
  direc = array[0..77] of direntry; { Directory mit 78 Eintraegen }

var directory:direc;
    unitno:integer;

procedure hole_unitno; { Fragen, von welcher Unit gelesen werden soll }
var ch:char;
    ok:boolean;
begin
  write('Von welcher Unit lesen (4/5) ?');
  ok:=false;
  repeat
    read(ch);
    if ch in ['4','5']
      then ok:=true
      else write(chr(7),chr(8),' ',chr(8));
  until ok;
  unitno:=ord(ch)-48;
end;

procedure lese_dir; { liest das Directory von der Unit #4 ein }
var io:integer;     { Bei einem Disketten-Fehler wird das Programm verlassen }
begin
  {$I-}
  unitread(unitno,directory,sizeof(directory),2);
  {$I+}
  io:=ioresult;
  if io<>0 then
    { Fehler ! }
    begin
      writeln(chr(7)); { Beep }
      writeln('Fehler #',io);
      writeln('Directory von Unit #',unitno,' nicht zu lesen');
      exit(program)
    end;
end;

procedure gebe_dir_aus;
var i:integer;
begin
  writeln(chr(12)); { Schirm loeschen }
  with directory[0] do
    begin { Informationen ueber Diskette ausgeben }
      { Diskettename }
      write('Diskettenname:''',diskvol,'''');
      { zuletzt benutzt am }
      writeln(' / Zuletzt benutzt am ',lastboot.day,'-',lastboot.month,
              '-',lastboot.year);
      { Anzahl der Bloecke }
      write('Anzahl der Bloecke: ',blockno);
      { Anzahl der Files }
      writeln(' / Anzahl der Files: ',fileno);
      writeln;
```

```
      end;
    i:=1;
    while i<=directory[0].fileno do
      begin
        if i mod 20 = 0 then
          { Bildschirmausgabe anhalten }
          begin
            write('Bitte <ret> zum Weitermachen');
            readln;
            writeln(chr(12)); { Bildschirm loeschen }
          end;
        with directory[i] do { Informationen der einzelnen Files ausgeben }
          begin
            { Filename }
            write(copy(concat(filename,'                  '),1,18));
            { belegte Bloecke }
            write('Block ',firstblock:3,'-',lastblock:3,'  ');
            write('(',lastblock-firstblock:3,') ');
            { geschrieben am }
            write(lastaccess.day:2,'-',lastaccess.month:2,'-',
                  lastaccess.year:2,'  ');
            { Filetyp }
            case filekind of
              bad : write('Bad File');
              code: write('Codefile');
              text: write('Textfile');
              info: write('Infofile');
              data: write('Datafile');
              graf: write('Graffile');
              foto: write('Fotofile')
            end;
            { Bytes im letzten Block }
            writeln('  ',endbytes:3);
          end;
        i:=i+1;
      end;
end;

begin
  hole_unitno;
  lese_dir;
  gebe_dir_aus;
end.
```

2.3 PASH: Ein alternativer Filer

Aus Kapitel 2.2 haben wir die Informationen, um das Directory einer jeden Diskette zu lesen und zu verändern. In PASH werden wir dieses Wissen ausgiebig anwenden.

Das Ändern der Informationen über eine Diskette ist im POS-Betriebssystem die Aufgabe des Filers. Fast sämtliche seiner Funktionen beruhen darauf. Das Verändern des Volume-Namens mit der "C"-Funktion ist im Grunde genommen nur das Verändern eines Feldes der Directory-Informationen. Da wir dies auch können, wollen wir nun ein Programm entwickeln, das einige Funktionen des Filers übernimmt und neue hinzufügt. Da das Programm an ein Programm angelehnt ist, das unter dem Betriebssystem CP/M läuft und WASH heißt, nennen wir unser Programm PASH.

Was ist am Filer noch zu verbessern, könnte man sich fragen, wenn man damit bis jetzt jedes auftauchendes Problem lösen konnte. Es fallen einem auch nur wenige Funktionen ein, die man ab und zu gebrauchen könnte, die der Filer aber nicht bietet.

So wäre es gut, wenn man einige Standardvorgänge mit einem Kommando auslösen könnte. So z.B. das Listen eines Textes auf dem Drukker. Falls man ein entsprechendes Kommando hat, braucht man nur noch einen Filenamen eingeben und nicht mehr, daß man den Text an den Drukker ("PRINTER:") schicken will (Mit "T"-Kommando).

Wir bauen also in unser Programm eine solche Funktion ein. Ebenfalls wird eine Funktion realisiert, die ein File auf dem Bildschirm ausgibt. Schließlich wäre es auch schön, wenn wir uns auch Codefiles anschauen könnten, ohne daß dies ein wildes Piepsen und Bildschirm löschen auslöst.

Da die meisten Pascalbenutzer mit zwei Laufwerken arbeiten, gibt es nur vier Richtungen zum Kopieren von Files, die benutzt werden. Also vereinfachen wir das Kopieren von Files, indem die Kopierrichtung vorher festgelegt wird. Dann ist nur noch ein Filename für ein Kopierkommando nötig.

Was wir in unserem Programm auch gegenüber dem Filer verbessern wollen, ist der Bedienungskomfort. Für fast alle Filerkommandos ist die Eingabe eines Filenamens nötig. Bei längeren Namen wird es aber schwierig, sich die genaue Schreibweise des Namens zu merken. Man muß sich also zuerst das Directory mit "L" listen lassen um den Namen dann abzuschreiben. Zudem wird der Bildschirm oft gelöscht, so daß wir uns das Direktory erneut listen müssen. Um dies zu umgehen, führen wir eine völlig andere Eingabe des Filenamens ein.

Der größte Teil des Bildschirms bei der Benutzung von PASH wird reserviert für die Anzeige aller Filenamen einer Diskette. Wenn wir uns nur auf den Filenamen beschänken paßt auch die maximale Anzahl von Files auf den Bildschirm. Nun benutzen wir noch einen Cursor, mit dem wir auf einen Filenamen zeigen können. Die Angabe eines Filenamens beschänkt sich damit nur auch einige Cursorbewegungen. Wenn dann ein Kommando eingegeben wird, bezieht es sich auf das File, auf das der Cursor gerade zeigt. Es braucht also nicht ein einziger Buchstabe des Filenamens eingegeben werden. Fehleingaben sind ausgeschlossen und man behält immer den Überblick über den Inhalt der ganzen Diskette, mit der gerade gearbeitet wird.

Um die Anzahl der Eingaben zur Cursorbewegungen zu minimieren führen wir Kommandos ein, die den Cursor ein oder fünf Files vor oder zurück bewegen. Zwei weitere ermöglichen es, zum ersten oder zum letzten Eintrag im Directory zu springen.

Schließlich führen wir noch ein Konzept ein. Die Auswahl von mehreren Files ist im Filer durch die sogenannten "Wildcards" "=" und "?" realisiert. Es ist aber nicht möglich, zwei Files automatisch hintereinander zu bearbeiten, wenn kein Teil ihrer Namen identisch ist. Das "?"-Zeichen, das für solche Fälle gedacht ist, erfordert wieder bei jedem File ein "J/N"-Antwort.

Hier führen wir nun Marken ein. Jedes File, auf dem der Cursor steht kann markiert werden. Wird nun ein Kommando aufgerufen, das z.B. alle markierten Files löscht, so geschieht das automatisch. Der Vorteil ist, daß mit einem Kommando Files bearbeitet werden können, die keinerlei Entsprechungen im Filenamen aufweisen müssen.

PASH bietet nicht alle Funktionen des Filers, da das Programm dann einen beträchtlichen Umfang erreichen würde, der den Rahmen eines Buches sprengt. PASH kann Files löschen, umbenennen, kopieren und auf dem Bildschirm und auf dem Drucker ausgeben.

Es ist für den Leser aber auch möglich, PASH zu erweitern. Dabei kann er die Cursorsteuerung und das Markenkonzept unberührt lassen. Er braucht sich nur die entsprechenden Prozeduren schreiben, und die mit einem bestimmten Tastaturkommando aufrufen lassen.

Damit dies erleichtert wird, schauen wir uns eine Funktion von PASH genauer an, undzwar die Löschfunktion.

PASH bietet zwei Kommandos zum Löschen von Files an. Das File, auf das der Cursor zeigt, kann gelöscht werden oder alle markierten Files. Beide benötigen ein Unterprogramm, das ein einzelnes File löscht. Diese Prozedur heißt "DELENTRY". An sie wird ein Integerwert übergeben, der die Nummer des Eintrages im Directory angibt. Für das erste File würde z.B. der Wert 1 übergeben.

Wenn überhaupt Files vorhanden sind, gibt "DELENTRY" eine Meldung aus, daß ein File gelöscht wird. Dann werden einfach alle Directory-Einträge, die hinter den zu löschenden File stehen um eins aufgerückt, da alle Einträge aufeinander folgen müssen. Dann wird nur noch der Directory-Eintrag "FILENO" um 1 erniedrigt. Er gibt an, wieviele Files auf der Diskette vorhanden sind. Damit ist das File schon gelöscht. Es ist ersichtlich, daß die Operationen mit dem Directory äußerst simpel sind.

Die Prozedur "DEL" löscht das File, auf das der Cursor gerade zeigt. Der Benutzer muß zunächst das Löschen durch "Y" bestätigen. Die Funktion "YESNO" erwartet die Eingabe von "Y" oder "N" und ergibt bei "Y" den Wert "TRUE".

Bei "Y" löscht "DEL" mit "DELENTRY" das File. Die Nummer des Files, auf das der Cursor zeigt steht in "CURRENT". Dann wird mit "PUTDIRECTORY" das Directory auf die Diskette geschrieben. "PUTDIRECTORY" ergibt den Wert "TRUE", wenn ein I/O-Fehler aufgetreten ist. Falls ja, gibt "DEL" eine Fehlermeldung aus und liest das Directory wieder neu ein.

Schließlich wird das Directory mit "PRINTDIR" auf dem Bildschirm neu ausgeben und der Cursor eventuell berichtig. Dies ist dann der Fall, wenn das letzte File gelöscht wurde, da er dann außerhalb der Einträge stehen würde.

"TAGDEL" löscht alle markierten Files. Zunächst wird gefragt, ob jede Aktion bestätigt werden soll, was in der Variablen "CONFIRM" festgehalten wird. Mit einer Schleife werden alle Files daraufhin geprüft, ob sie markiert sind. Dies ist der Fall, wenn das Feld "TAGGED" für den Eintrag den Wert "TRUE" hat. Dann wird eventuell nachgefragt, ob wirklich gelöscht werden soll und mit "DELENTRY" das File gelöscht.

Daraufhin wird wieder der Cursor korrigiert und das Directory auf die Diskette geschrieben. Falls ein Fehler auftritt, gibt es wieder eine Meldung. Schließlich wird das Directory wieder neu auf dem Bildschirm aufgezeigt.

Wichtige Typen und Variablen

"DIREC" : Enthält einen strukturierten Record für das Directory. Wird aus Kapitel 2.2 übernommen.

"DIRECTORY" : Enthält das Directory der Diskette, mit der gerade gearbeitet wird.

"TAGGED" : Feld, das angibt, ob ein File markiert ist. Bei "TRUE" ist der entsprechende Directory-Eintrag markiert.

"CURRENT" : Enthält die Nummer des Files, auf das der Cursor im Moment zeigt.

"SOURCE" : Enthält Unitnummer des Laufwerks, mit dem gerade gearbeitet wird (Unit 4 oder 5). Beim Kopieren wird von dieser Unit gelesen.

"TARGET" : Enthält Unitnummer des Laufwerks, auf das beim Kopieren geschrieben wird (Unit 4 oder 5).

"JMPDIR" : Enthält die Richtung, in der das "J"-Kommando arbeitet (1 oder -1).

Kommandoliste

"->" : Cursor einen Eintrag nach unten bewegen
"<-" : Cursor einen Eintrag nach oben bewegen
"B" : Cursor auf ersten Eintrag setzen
"E" : Cursor auf letzten Eintrag setzen
"J" : Cursor fünf Einträge nach oben oden nach unten bewegen
"," : Richtung für "J" Kommando auf "nach oben" setzen
"." : Richtung für "J" Kommando auf "nach unten" setzen

<spc>: Marke setzen oder löschen
"4" : von Unit 4 lesen, auf Unit 5 kopieren
"$" : von Unit 4 lesen, auf Unit 4 kopieren
"5" : von Unit 5 lesen, auf Unit 4 kopieren
"%" : von Unit 5 lesen, auf Unit 5 kopieren

"S" : Directory neu einlesen
"D" : File löschen (Muß durch "Y" bestätigt werden)
"F" : markierte Files löschen (durch "Y" wird ausgewählt, daß bei jedem File bestätigt werden muß)
"R" : File umbenennen
"C" : File kopieren (Falls vorhanden, muß Überschreiben durch "Y" bestätigt werden)
"V" : markierte Files kopieren
"L" : File auf dem Bildschirm ausgeben (Durch "Y" kann ausgewählt werden, daß nach jeder Bildschirmseite durch <esc> abgebrochen werden kann)
"P" : File auf Drucker ausgeben
"I" : Fileinformationen ausgeben (Reihenfolge: Filename/Datum/belegte Blöcke/Bytes im letzten Block)
"U" : ausgeben, wieviel Platz auf der Diskette schon belegt und noch frei ist (Angaben in Blöcken und KByte)

    "/" : Kommandoliste ausgeben
    "X" : PASH verlassen

Programm "PASH.TEXT"

```
{$S+}
programm PASH;
const
  window=20;
type
  setofchar = set of char;
  ab = (a,b);
  vname=string[7];
  fname=string[15];
  DATEREC = PACKED RECORD
              MONTH: 0..12;
              DAY:   0..31;
              YEAR:  0..100;
            END;
  FKIND = (vol,bad,code,text,info,data,graf,foto,secr);
  DIRENTRY = RECORD
               FIRSTBLOCK: INTEGER;
               LASTBLOCK: INTEGER;
               CASE FILEKIND: FKIND OF
                    vol,secr: (diskvol: VNAME;
                               blockno: INTEGER;
                               fileno:integer;
                               accesstime: INTEGER;
                               lastboot: DATEREC);
                    bad,code,text,
                    info,data,graf,foto: (FILENAME: FNAME;
                                          ENDBYTES: 1..512;
                                          lastaccess: DATEREC);
            END;
  DIREC = ARRAY[0..77] OF DIRENTRY;

var
  cr,bs,sp,eol,bell : char;
  directory:direc;
  tagged: packed array[1..77] of boolean;
  currentold,current:integer;
  cmdset:setofchar;
  noclr:boolean;
  source,target:integer;
  sources,targets:string[3];
  jmpdir:integer;

segment procedure init; { initialisiert Variablen und legt das }
begin                   { Bildschirmlayout fest }
  page(output);
  write('-----------------------------------');
  write(' PASH 1.1 ');
  write('-----------------------------------');
  gotoxy(0,21);
  write('-Read on <4>--Copy <4 -> 5>--Jump<+>----');
  write('----------------------------------------');
  cr:=chr(13);
  bs:=chr(8);
  sp:=chr(21);
  eol:=chr(29);
```

```
  bell:=chr(7);
  cmdset:=[bs,sp,' ','$','%',',','.','/','4','5','B','C','D','E','F','I','J',
          'L','P','R','S','U','V','X'];
  current:=1;
  currentold:=1;
  noclr:=false;
  fillchar(tagged,sizeof(tagged),chr(0));
  source :=4;      target :=5;
  sources:='#4:'; targets:='#5:';
  jmpdir:=1;
end;

procedure cleartags; { loescht alle Markierungen }
begin
  fillchar(tagged,sizeof(tagged),chr(0));
end;

function getchar(okset:setofchar):char; { holt ein Zeichen von der Tastatur, }
var                                      { das in okset entahlten sein muss   }
  ch   : char;
  good : boolean;
begin
  repeat
    read(keyboard,ch);
    if ord(ch)>95 then ch:=chr(ord(ch)-32);
    if eoln(keyboard) then ch:=cr;
    good := ch in okset;
    if not good then write(bell)
      else if ch in [' '..chr(125)] then write(ch);
  until good;
  getchar:=ch;
end;

procedure getstring(var s:string; okset:setofchar; maxlen:integer); { holt }
var                          { einen String von der Tastatur. Seine Laenge steht }
  s1      :string[1]; { in maxlen, die Zeichen muessen in okset enthalten }
  stemp   :string;    { sein. }
  len     :integer;
  firstchar,lastchar:boolean;
  getset  :setofchar;
begin
  s1:=' '; stemp:='';
  if maxlen<1 then maxlen:=1
    else if maxlen>255 then maxlen:=255;
  repeat
    len:=length(stemp);
    firstchar:=(len=0);
    lastchar:=(len=maxlen);
    if firstchar
      then getset:=okset
      else if lastchar then getset:=[cr,bs]
                       else getset:=okset+[cr,bs];
    s1[1]:=getchar(getset);
    if s1[1] in okset
      then stemp:=concat(stemp,s1)
      else if s1[1]=bs then begin
                              write(bs,' ',bs);
                              delete(stemp,len,1);
                            end;
  until s1[1]=cr;
  writeln;
```

```
    s:=stemp;
  end;

  procedure getaname(var name:string; var num:integer); { liest einen Filenamen }
  begin { von der Tastatur ein und prueft, ob er im Directory vorkommt. Falls    }
    getstring(name,[' '..'^'],15);                             { nicht, wird num 0 }
    num:=0;
    repeat
      num:=num+1;
    until (name=directory[num].filename) or
          (num>directory[0].fileno);
    if num>directory[0].fileno then num:=0;
  end;

  function yesno:boolean; { liest eine Y/N-Antwort ein }
  begin
    yesno:=(getchar(['N','n','Y','y']) in ['Y','y']);
  end;

  procedure ret; { wartet auf die Eingabe von <ret> }
  var dummy:char;
  begin
    write (' <ret>');
    dummy:=getchar([chr(13)]);
  end;

  function getdirectory:integer; { liest das Directory ein. Der Funktionswert }
  begin                          { entspricht dem I/O-Result. }
    (*$I-*)
    unitread(source,directory,sizeof(directory),2);
    (*$I+*)
    getdirectory:=ioresult;
  end;

  function putdirectory:integer; { schreibt das Directory auf die Diskette. }
  begin                        { Der Funktionswert entspricht dem I/O-Result. }
    (*$I-*)
    unitwrite(source,directory,sizeof(directory),2);
    (*$I+*)
    putdirectory:=ioresult;
  end;

  procedure error; { gibt Fehlermeldung aus }
  begin
    gotoxy(0,22);
    write(bell,'*** I/O error *** ');
    ret;
  end;

  procedure printentry(num:integer); { gibt einen Filenamen auf dem Schirm aus }
  var ch:char;
  begin
    gotoxy(1+19*trunc(num/(window+1)),1 + ((num-1) mod window));
    if tagged[num]
      then ch:='+'
      else ch:=':';
    write(ch,directory[num].filename);
    gotoxy(17+19*trunc(num/(window+1)),1 + ((num-1) mod window));
    write(ch);
  end;
```

```
procedure clear(num:integer); { loescht einen Filenamen auf dem Bildschirm }
begin
  gotoxy(2+19*trunc(num/(window+1)),1 + ((num-1) mod window));
  write('                 ');
end;

procedure clearwindow; { loescht Bildschirm }
var i:integer;
begin
  gotoxy(0,1);
  for i:=1 to window do
    writeln(eol);
  gotoxy(0,1);
end;

procedure clearline; { loescht die 22. Bildschirmzeile }
begin
  gotoxy(0,22);
  write(eol);
end;

procedure moncurrent; { gibt Cursor auf dem Bildschirm aus }
begin
  gotoxy(19*trunc(currentold/(window+1)),1 + ((currentold-1) mod window));
  write(' ');
  gotoxy(18+19*trunc(currentold/(window+1)),1 + ((currentold-1) mod window));
  write(' ');
  gotoxy(19*trunc(current/(window+1)),1 + ((current-1) mod window));
  write('>');
  gotoxy(18+19*trunc(current/(window+1)),1 + ((current-1) mod window));
  write('<');
  currentold:=current;
end;

procedure printdir; { gibt das Directory auf dem Bildschirm aus }
var i:integer;
begin
  clearwindow;
  for i:= 1 to directory[0].fileno do
    printentry(i);
  moncurrent;
end;

procedure helpuser; { gibt die Kommandoliste aus }

  procedure help1;
  begin
    clearwindow;
    writeln('                                    Commands',cr);
    write(' -> Move cursor down                        ');
    writeln('  J  Move cursor five up or down');
    write(' <-  Move cursor up                         ');
    writeln('  ,  Set J-direction to up');
    write('  B  Set cursor to first file               ');
    writeln('  .  Set J-direction to down');
    writeln('  E  Set cursor to last file');
    write(' spc Toggle tag                             ');
    writeln('  4: read from unit 4  copy to unit 5');
    write('  S  Start over                             ');
    writeln('  $: read from unit 4  copy to unit 4');
    write('  D  Delete file                            ');
```

```
    writeln('  5: read from unit 5  copy to unit 4');
    write('  F  Tagged delete                    ');
    writeln('  %: read from unit 5  copy to unit 5');
  end;

  procedure help2;
  begin
    writeln('  R  Rename file');
    writeln('  C  Copy file');
    writeln('  V  Tagged copy');
    writeln('  L  List a file on CRT');
    writeln('  P  List a file on PRINTER');
    write('  I  Print file information             ');
    writeln('  /  Print helptable');
    write('  U  Print used disk space              ');
    writeln('  X  Exit PASH');
    gotoxy(0,22);
    write('Press');
    ret;
    printdir;
  end;

begin
  help1;  { wegen der beschraenkten Groesse einer Prozedur musste }
  help2;  { 'help' aufgeteilt werden }
end;

procedure advance; { bewegt Cursor um einen Filennamen nach vorne }
begin
  if current<directory[0].fileno
    then begin
           current:=current+1;
           moncurrent;
           noclr:=true;
         end;
end;

procedure stepback; { bewegt Cursor um einen Filenamen zurueck }
begin
  if current>1
    then begin
           current:=current-1;
           moncurrent;
           noclr:=true;
         end;
end;

procedure jump;  { bewegt Cursor um 5 Filenamen vor oder zurueck }
var i:integer;
begin
  for i:=1 to 5 do
    case jmpdir of
     -1: stepback;
      1: advance
    end;
end;

procedure tag;  { setzt eine Markierung }
begin
  tagged[current]:=not tagged[current];
  printentry(current);
```

```
  advance;
  noclr:=false;
end;

procedure jmpbeg; { setzt den Cursor auf den ersten Filenamen }
begin
  current:=1;
  moncurrent;
  noclr:=true;
end;

procedure jmpend; { setzt den Cursor auf den letzten Filenamen }
begin
  current:=directory[0].fileno;
  moncurrent;
  noclr:=true;
end;

procedure restart(s:boolean); { liest Directory neu ein }
begin
  clearline;
  if s then write('Start over');
  if getdirectory <> 0
    then error
    else begin
           cleartags;
           printdir;
           jmpbeg;
           noclr:=false;
         end;
end;

procedure rename; { veraendert einen Filenamen }
var target:string;
 dummy,num:integer;
begin
  gotoxy(0,22);
  write('Rename as ');
  getaname(target,num);
  if num=0
    then begin
           directory[current].filename:=target;
           if putdirectory<>0 then begin
                                     error;
                                     dummy:=getdirectory;
                                   end;
           clear(current);
           printentry(current);
         end
    else begin
           gotoxy(0,22);
           write(target,' already exists !',eol);
         end;
end;

procedure filesize; { gibt Informationen zu einem File aus }
begin
  gotoxy(0,22);
  with directory[current] do
    begin
      write(filename,'  ');
```

```
        with lastaccess do
          write(day,'-',month,'-',year);
        write('  Used blocks: ',firstblock,'-',lastblock);
        write('  Bytes in last block: ',endbytes,' ');
      end;
    ret;
end;

procedure unused; { gibt aus, wieviel Platz auf der Diskette belegt ist }
var unused,used,i:integer;
begin
  gotoxy(0,22);
  write('Unused blks : ');
  used:=directory[0].lastblock-directory[0].firstblock;
  for i:= 1 to directory[0].fileno do
    used:=used+(directory[i].lastblock-directory[i].firstblock);
  unused:=directory[0].blockno-used;
  write(unused,' (= ',trunc(unused/2),'.',
        trunc((unused*10)/2-10*trunc(unused/2)),' K Bytes)');
  write(' / ',used,' blks used');
  write(' (= ',trunc(used/2),'.',
        trunc((used*10)/2-10*trunc(used/2)),' K Bytes) ');
  ret;
end;

procedure delentry(num:integer); { loescht ein File }
var i:integer;
begin
  if directory[0].fileno <> 0 then
    begin
      clearline;
      write('Deleting ',directory[num].filename);
      with directory[0] do
        begin
          if num<fileno then for i:= num to fileno-1 do
                                 directory[i]:=directory[i+1];
          fileno:=fileno-1;
        end;
    end;
end;

procedure del; { loescht ein File auf Benuztereingabe }
var dummy:integer;
begin
  clearline;
  write('Delete ? (Y/N) ');
  if yesno then begin
                  delentry(current);
                  if putdirectory<>0 then begin
                                             error;
                                             dummy:=getdirectory;
                                           end;
                  printdir;
                  if current>directory[0].fileno then
                    current:=directory[0].fileno;
                  moncurrent;
                end;
end;

procedure tagdel; { loescht alle markierten Files }
var dlt,confirm:boolean;
```

```
        dummy,i:integer;
begin
  gotoxy(0,22);
  write('Comfirm each delete ? (Y/N) ');
  confirm:=yesno;
  for i := directory[0].fileno downto 1 do
    if tagged[i] then begin
                        if confirm
                          then begin
                                 clearline;
                                 write(directory[i].filename,'  Delete ? (Y/N) ');
                                 dlt:=yesno;
                               end
                          else dlt:=true;
                        if dlt then delentry(i);
                      end;
  cleartags;
  if current>directory[0].fileno then current:=directory[0].fileno;
  if putdirectory<>0 then begin
                            error;
                            dummy:=getdirectory;
                          end;
  printdir;
end;

procedure view(list:boolean); { gibt ein File auf dem Bildschirm }
var i,j,                      { oder dem Drucker aus. }
k,count,
  dummy: integer;
 carray: packed array[0..15] of
           packed array[0..63] of char;
  unfil: file;
  outpt: interactive;
newline,
  abort,
    dle: boolean;

procedure chkio; { verlaesst 'view' bei Diskettenfehler }
begin
  if ioresult<> 0 then begin
                         error;
                         printdir;
                         exit(view)
                       end;
end;

procedure askuser; { gibt Bildschirmmeldung aus }
var an:char;
begin
  clearline;
  write('Press <ret> or <esc> ');
  an:=getchar([cr,chr(27)]);
  clearline;
  if an=chr(27) then begin
                       close(unfil);
                       printdir;
                       exit(view)
                     end;
end;

function readblk:integer; { liest einen Block des Files }
```

```
begin
  (*$I-*)
  readblk:=blockread(unfil,carray,2);
  (*$I+*)
  chkio;
end;

begin
  gotoxy(0,22);
  write('Do you want the possibility to abort ? (Y/N)');
  abort:=yesno;
  clearline;
  clearwindow;
  (*$I-*)
  reset(unfil,concat(sources,directory[current].filename));
  if list then reset(outpt,'PRINTER:')
          else reset(outpt,'CONSOLE:');
  (*$I+*)
  chkio;
  if directory[current].filekind = text
    then
      begin
        dummy:=readblk;
        count:=0;
        newline:=true;
        dle:=false;
        repeat
          for i:= 0 to readblk*7+1 do
            begin
              for j:= 0 to 63 do
                begin
                  if dle then begin
                                for k:=1 to ord(carray[i,j])-32 do
                                  write(outpt,' ');
                                dle:=false;
                              end
                         else
                              if newline then
                                                  if carray[i,j]=chr(16)
                                                    then dle:=true
                                                    else begin
                                                           newline:=false;
                                                           if carray[i,j]=chr(13)
                                                             then
                                                               begin
                                                                 writeln(outpt);
                                                                 newline:=true;
                                                                 count:=count+1;
                                                               end
                                                             else
                                                               write(outpt,carray[i,j]);
                                                         end
                                              else
                                                if carray[i,j]=chr(13)
                                                  then begin
                                                         writeln(outpt);
                                                         newline:=true;
                                                         count:=count+1;
                                                       end
                                                  else write(outpt,carray[i,j]);
                 if count=20 then begin
```

```
                                          count:=0;
                                          if abort then askuser;
                                          if not list then clearwindow;
                                        end;
                   end;
               end;
           until eof(unfil);
         end
       else begin
              repeat
                if not list then clearwindow;
                for i:= 0 to readblk*7+1 do
                  begin
                    for j:= 0 to 63 do
                      if (carray[i,j] < chr(32))
                         or ((carray[i,j]>chr(126)) and (carray[i,j]<chr(160)))
                         or (carray[i,j]=chr(255))
                        then write(outpt,'.')
                        else write(outpt,carray[i,j]);
                    writeln(outpt);
                  end;
                if not eof(unfil) then
                  if abort then askuser;
              until eof(unfil);
            end;
  (*$I-*)
  close(unfil);
  close(outpt);
  (*$I+*)
  clearline;
  write('Press');
  ret;
  printdir;
end;

function copyfile(num:integer):boolean; { kopiert ein File }
var filea,fileb:file;
    change,repl:boolean;
    dirb:direc;
    blks,i:integer;
    blkarray: packed array[0..24063] of char;

procedure bye; { verlaesst 'copyfile' }
begin
  (*$I-*)
  close(filea);
  close(fileb);
  (*$I+*)
  copyfile:=true;
  exit(copyfile)
end;

procedure chkio; { testet auf Diskettenfehler }
var hold:integer;
begin
  hold:=ioresult;
  if hold=8 then begin
                   clearline;
                   write('No room on copy disk. ');
                   ret;
                   bye;
```

```
                  end;
   if hold <> 0 then begin
                       error;
                       bye;
                     end;
end;

procedure prompt(which:ab); { gibt Bildschirmmeldung aus }
begin
  gotoxy(0,22);
  write(eol,'Insert ');
  case which of
    a: write('original');
    b: write('copy')
  end;
  write(' disk in drive #',source,' ');
  ret;
  clearline;
end;

begin
  copyfile:=false;
  gotoxy(0,22);
  write(eol,'Copying ''',directory[num].filename,'''');
  change:=(source=target);
  (*$I-*)
  reset(filea,concat(sources,directory[num].filename));
  chkio;
  if change then prompt(b);
  unitread(target,dirb,sizeof(dirb),2);
  chkio;
  for i:=1 to dirb[0].fileno do
    if dirb[i].filename=directory[num].filename then
      begin
        i:=dirb[0].fileno+1;
        gotoxy(0,22);
        write(eol,'File exists. Replace ? (Y/N) ');
        repl:=yesno;
        clearline;
        if not repl then bye;
      end;
  rewrite(fileb,concat(targets,directory[num].filename));
  chkio;
  repeat
    if change then prompt(a);
    blks:=blockread(filea,blkarray,47);
    chkio;
    if change then prompt(b);
    blks:=blockwrite(fileb,blkarray,blks);
    chkio;
  until eof(filea);
  close(fileb,lock);
  chkio;
  if change then prompt(a);
  close(filea);
  (*$I+*)
  chkio;
  if getdirectory <> 0
    then error
    else begin
           printdir;
```

```
          jmpbeg;
          noclr:=false;
        end;
end;

procedure copy; { kopiert File auf Benutzereingabe }
var dummy:boolean;
begin
  write('Copying');
  dummy:=copyfile(current);
  clearline;
end;

procedure tagcopy; { kopiert alle markierten Files }
var i:integer;
begin
  gotoxy(0,22);
  clearline;
  for i:=1 to directory[0].fileno do
    if tagged[i] then
      if copyfile(i) then begin
                             clearline;
                             write('Do you want to continue ? (Y/N) ');
                             if not yesno then exit(tagcopy)
                                          else clearline;
                           end;
  cleartags;
  printdir;
end;

procedure setunit(which:integer; same:boolean); { setzt Kopierrichtung }
var oldsource:integer;
begin
  oldsource:=source;
  source:=which;
  case source of
    4: target:=5;
    5: target:=4
  end;
  if same then target:=source;
  case source of
    4: sources:='#4:';
    5: sources:='#5:'
  end;
  case target of
    4: targets:='#4:';
    5: targets:='#5:'
  end;
  gotoxy(10,21);
  write(source);
  gotoxy(20,21);
  write(source,' -> ',target);
  if oldsource<>source then restart(false);
end;

procedure setjmpdir(dir:integer); { setzt Richtung fuer das 'J' Kommando }
begin
  jmpdir:=dir;
  gotoxy(34,21);
  case jmpdir of
   -1: write('-');
```

```
    1: write('+')
  end;
end;

begin
  init;
  if getdirectory <> 0 then begin
                              error;
                              exit(program)
                            end;
  printdir;
  repeat { endlose Hauptschleife }
    if not noclr then clearline
                 else noclr:=false;
    gotoxy(0,22);
    write('> ',bs);
    case ord(getchar(cmdset)) of
       8: stepback;          (* <-- *)
      21: advance;           (* --> *)
      32: tag;               (* ' ' *)
      36: setunit(4,true);   (* '$' *)
      37: setunit(5,true);   (* '%' *)
      44: setjmpdir(-1);     (* ',' *)
      46: setjmpdir(1);      (* '.' *)
      47: helpuser;          (* '/' *)
      52: setunit(4,false);  (* '4' *)
      53: setunit(5,false);  (* '5' *)
      66: jmpbeg;            (* 'B' *)
      67: copy;              (* 'C' *)
      68: del;               (* 'D' *)
      69: jmpend;            (* 'E' *)
      70: tagdel;            (* 'F' *)
      73: filesize;          (* 'I' *)
      74: jump;              (* 'J' *)
      76: view(false);       (* 'L' *)
      80: view(true);        (* 'P' *)
      82: rename;            (* 'R' *)
      83: restart(true);     (* 'S' *)
      85: unused;            (* 'U' *)
      86: tagcopy;           (* 'V' *)
      88: begin              (* 'X' *)
            write(chr(12));
            exit(program)  { beendet das Programm }
          end
    end;
  until false;
end.
```

2.4 Kein Absturz bei segmentierten Programmen

Apple-Pascal bietet die Möglichkeit, beliebig lange Programm abzuarbeiten. Dazu werden einzelne Programmteile in sog. "Segmenten" zusammengefasst. Beim Aufruf eines Segments wird der entsprechende Teil von der Diskette nachgeladen. Wird er nicht mehr gebraucht, kann an seiner Stelle ein anderes Segment geladen werden. Dadurch kann ein Programm größer als der vorhandene Speicherplatz werden, da ja immer nur Teile davon im Rechner stehen müssen. Die Realisierung der Segmentierung im Programm wird im "Apple Pascal Language" Hand buch auf der Seite 74 beschrieben.

Diese Verfahren birgt jedoch eine Gefahr. Das zu ladende Segment ist ein Teil des Programmfiles. Deshalb kann es auch nur von der Diskette geladen werden, auf der das Programmfile steht. Apple-Pascal merkt sich nur, in welchen Blöcken dieses Segment auf der Diskette steht. Es wird aber nicht geprüft, ob sich im Laufwerk die richtige Diskette befindet. Ist es eine andere, so stürzt der Rechner unweigerlich ab. Die Folgen sind vor allem Datenverlust.

Ein segmentiertes Programm, an dem wir diesen Effekt nachprüfen können, ist der Editor. Er besteht aus mehreren Segmenten, so z.B. eines für das Ausführen des "CF" Befehls. Damit wird zu dem im Speicher befindlichen Text ein anderer von Diskette dazugeladen. Wir starten also den Editor und geben dann "CF" ein. Nun wird das entsprechende Segment geladen, was man an der Laufwerksleuchte merkt. Es folgt die Frage nach einem Filenamen. Wir legen nun eine andere Diskette in das Laufwerk und geben den Namen eines Textfiles ein, das sich auf ihr befindet. Nun wird der Text nachgeladen. Dann erscheint die Meldung, das wir nachprüfen sollen, ob sich das File "SYSTEM.EDITOR" im Laufwerk befindet. Wir wechseln nun aber nicht die Diskette, sondern drücken <ret>. Das Ergebnis ist, daß nun ein anderes Segment eingeladen werden soll. Nun werden aber einfach bestimmte Blöcke von der Diskette, die sich gerade im Laufwerk befindet geladen. Da dies nicht die richtige ist, werden sinnlose Daten geladen. Vielleicht erscheinen auf dem Bildschirm einige seltsame Zeichen, auf jeden Fall ist ein Weiterarbeiten unmöglich, da der Rechner abgestürzt ist.

Diese Fehlbedienung wollen wir in diesem Kapitel abfangen lernen. Nachdem wir in Kap. 2.2 den Aufbau des Directorys kennengelernt haben, können wir von einem Programm aus feststellen, welchen Namen die Diskette hat, die sich gerade in einem Laufwerk befindet. Dies nützen wir hier aus.

Wir gehen dabei so vor, daß wir zu Beginn des Programms feststellen, welche Diskette sich im Laufwerk befindet. Da dieser Teil am Angang des Programms steht, ist es nicht sehr wahrscheinlich, das inzwischen die Diskette gewechselt wurde. Den Namen, den wir erhalten, bezeichnet die Programmdiskette. Sie muß sich immer dann im Laufwerk befinden, wenn ein Segement aufgerufen, oder verlassen wird.

Dies festzustellen ist ebenfalls einfach. Wir lesen nur wieder den Namen der Diskette ein, und vergleichen ihn mit dem, den wir an Anfang des Programms als Namen der Programmdiskette eingelesen hatten. Stimmen sie überein, so können wir ein Segment aufrufen, oder es verlassen. Ansonsten fordern wir den Benutzer auf, die richtige Diskette einzulegen und prüfen erneut.

Um einen Diskettennamen zu erhalten, brauchen wir aber nicht das gesamte Directory einzulesen. Wenn wir uns seinen Aufbau nochmal genau anschauen, werden wir festellen, das vor dem Namen der Diskette in 3 Wörtern andere Informationen stehen, die uns aber nicht interessieren. Wir fassen sie als ein Integerfeld mit drei Elementen zusammen. Dann folgt der Name der Diskette, ein String mit der Länge 7. Die nachfolgenden Informationen brauchen wir nicht einzulesen.

Nun zu dem hier abgedruckten Programm. Zu Beginn des Programmablaufs lesen wir in dem Segment "INIT" den Namen der Diskette ein, die sich gerade im Laufwerk befindet. Er wird an die globale Veriable "SEGSTRING" übergeben. Damit weiß das Programm, von welcher Diskette Segmente zu laden sind.

Dies wenden wir in dem Segment "SEG1" an. Die Funktion "CHKSEG" ergibt ein "TRUE", wenn sich die Diskette mit dem angegeben Diskettennamen im Laufwerk befindet. Sie wird in einer "REPEAT"-"UNTIL"-Schleife aufgerufen, solange, bis sie "TRUE" ergibt. Dann kann das Segment verlassen werden.

"GETSEGSTR" und "CHKSEG" übergeben bei einem Diskettenfehler den Fehlercode. Im abgedruckten Beispiel wird darauf das Programm abgebrochen.

Vielleicht fällt es auf, daß im dem Programm nicht geprüft wird, ob sich die Programmdiskette auch beim Aufruf eines Segments im Laufwerk befindet. Dies ist hier auch nicht nötig, da die Segmente direkt hintereinander aufgerufen werden. Es ist auszuschließen, daß in dieser kurzen Zeit die Diskette gewechselt werden könnte.

Wichtige Konstanten, Typen und Variable

"UNITNO" : Konstante, die angibt, von welcher Unit das Programm gestartet werden muß. Hier Unit #4

"DIRPART" : Record, um den Namen einer Diskette einzulesen. "DUMMY" enthält für uns unwichtige Informationen.

"SEGSTRING" : String, der den Namen der Pogrammdiskette aufnimmt.

Wichtige Prozeduren und Funktionen

"GETSEGSTRING" : Liest den Namen der Programmdiskette und übergibt einen eventuellen Fehlercode.

"CHKSEG" : Überprüft, ob sich die Programmdiskette im Laufwerk befindet. Ergibt "TRUE", wenn ja. übergibt einen eventuellen Fehlercode.

Programm "SEGCHK.TEXT"

```
programm seg;

const unitno = 4;

type
  vname=string[7];  { Ein Volumename }
  dir_part = record { ein Teil des Directorys }
                dummy: packed array [0..2] of integer;
              diskvol: vname;    { Name der Diskette }
             end;

var segstring : vname;

segment procedure init;
var error:integer;

  procedure get_segstr(var s:vname; var er:integer); { Namen der Programm- }
  var prt:dir_part;                                  { diskette einlesen   }
  begin
    {$I-}
```

```
    unitread(unitno,prt,sizeof(prt),2);
    {$I+}
    er:=ioresult;   { ev. Fehlercode retten }
    s:=prt.diskvol; { s setzen }
  end;

begin
  writeln('Segment INIT');  { Demo Meldung }

  get_segstr(segstring,error); { "segstring" von Diskette lesen }
  if error<>0 then begin         { bei Fehler Programm verlassen  }
                     writeln('Diskfehler');
                     exit(program);
                   end;

  { hier koennen weiter Anweisungen stehen }
  { . }
  { . }
  { . }

end;

segment procedure seg1;
var error:integer;

  function chkseg(s:string; var er:integer):boolean; { ergibt true, wenn }
  var prt:dir_part;          { die eingelegte Diskette den angegebenen Namen  }
  begin                      { hat, sonst false. Bei einem Fehler wird der    }
    {$I-}                    { Fehlercode uebergeben, und chkseg ergibt true  }
    unitread(unitno,prt,sizeof(prt),2);
    {$I+}
    er:=ioresult; { ev. Fehlercode retten }
    if prt.diskvol<>s then chkseg:=false
                      else chkseg:=true;
    if er<>0 then chkseg:=true;  { Korrektur bei Lesefehler }
  end;

begin
  writeln('Segment SEG1'); { Demomeldung }

  { hier koennen weiter Anweisungen stehen }
  { . }
  { . }
  { . }

  repeat  { Darauf warten, dass die richtig Diskette eingelegt wird }
    write('Bitte Programmdiskette einlegen');
    readln;
  until chkseg(segstring,error);
  if error<>0 then begin  { Bei Fehler Programm verlassen }
                     writeln('Diskfehler');
                     exit(program)
                   end;
end;

begin { Nur Segmente aufrufen }
  init;
  seg1;
end.
```

2.5 Wie man den DOS 3.3 Catalog liest

Das Standard-Betriebssystem des Apples ist das DOS, in der Version 3.3. Es wird mit jedem Diskettenlaufwerk ausgeliefert. Die Struktur einer DOS-Diskette ist der einer Pascal-Diskette in gewissen Maße ähnlich. Es bestehen jedoch mehrere Unterschiede.

So beträgt unter Pascal die Größe eines Block nicht wie unter POS 512, sondern 256 Bytes. Das entspricht einem halben Block. Er wird Sektor genannt. Außer diesem Hauptunterschied ist der Aufbau und die Position des Inhaltsverzeichnisses der Diskette, das unter DOS "Catalog" genannt wird, grundsätzlich verschieden.

Weiterhin ist die Position der Blöcke und Sektoren anders. Es ist also nicht so, daß z.B. der Block Nr. 1 dem zweiten und dritten Sektor vom Track 0 einer DOS-Diskette entspricht. Die genaue Verteilung ist in der Tabelle 2.5 für einen Track angegeben.

DOS-Sektor	entspricht POS-Block
0	0 linke Hälfte
1	7 linke Hälfte
2	6 rechte Hälfte
3	6 linke Hälfte
4	5 rechte Hälfte
5	5 linke Hälfte
6	4 rechte Hälfte
7	4 linke Hälfte
8	3 rechte Hälfte
9	3 linke Hälfte
10	2 rechte Hälfte
11	2 linke Hälfte
12	1 rechte Hälfte
13	1 linke Hälfte
14	0 rechte Hälfte
15	7 rechte Hälfte

Tabelle 2.5: Umrechnungstabelle DOS-Sektoren -> Pos-Blöcke

Jedoch sind POS- und DOS-Diskette nach der gleichen Methode formatiert, so das es möglich ist, unter Pascal Informationen einer DOS-Diskette zu lesen. Hier nützen einem Befehle wie "RESET", "GET" oder "BLOCKREAD" freilich nichts. Man ist auf die "UNITREAD/WRITE" Befehle beschränkt. Mit ihnen kann man einzelne Byte-Blöcke von der Diskette lesen, wobei die absolute Position dieser Bytes auf der Diskette angegeben wird. Es ist damit möglich, unter Pascal DOS-Files einzulesen, oder zu schreiben. In diesem Kapitel wird ein Programm vorgestellt, daß das sog. VTOC und den Catalog einer DOS-Diskette einliest und ausgibt.

Die Informationen über den Aufbau des DOS-Inhaltsverzeichnisses stehen im "DOS Manual" auf den Seiten 129-134.

Wir wollen mit dem Aufbau des sog. "VTOC" beginnen. "VTOC" bedeutet "Volume Table Of Contents". Es enthält Informationen darüber, wo der Catalog beginnt, allgemeine Informationen über den Aufbau der Dis-

kette und eine Liste, welche Sektoren belegt, und welche frei sind. Normale DOS-Disketten unterscheiden sich hier nur in dieser Liste. Wir wollen das VTOC als einen gepackten Record darstellen. Diesen könne wir dann direkt mit "UNITREAD" einlesen. Der Leser sollte nun anhand des DOS-Handbuchs auf Seite 132 den Aufbau des Records verfolgen.

Da die Informationen hier in einzelnen Bytes enthalten sind, definieren wir im Programm zunächst den Typ "BYTE", mit dem Wertbereich von 0 bis 255.

Das erste Byte im VTOC ist unbenutzt, wir nennen es "UNUSED1". Das zweite gibt an, auf welchem Track sich der erste Sektor des Catalogs befindet. Wir nennen es "CATTRK". Das dritte gibt an, bei welchem Sektor er beginnt. Es heißt "CATSEC". Byte 4 enthält die Nummer der DOS-Version ("DOSVERS"). Dann folgen zwei unbenutzte Bytes, die wir als ein gepackes Feld von zwei Byte auffassen. Darauf kommt die Volume-Nummer der Diskette. Wir nennen sie "VOLUME".

Die Bytes 7 bis 38 sind wieder unbenutzt. Eigentlich könnte man sie wieder als ein gepacked Feld mit 30 Bytes auffassen. Dies wird jedoch zu Fehlinformationen führen. Der Grund ist, das innerhalb eines gepackten Records gepackte Felder bei einem Byte mit gerader Stellenzahl beginnen müssen. Dies liegt daran, daß der interne P-Code-Interpreter mit 16-Bit-Werten rechnet. In unserem Fall hätte das zur Folge, daß ein gepackes Feld hier bei Byte Nr. 8 beginnen würde. Das erste unbenutzte Byte steht hier jedoch schon bei Nr. 7. Alle nachfolgenden Informationen wären also um ein Byte verschoben, und damit nicht mehr richtig. Wir müssen uns hier damit helfen, daß wir zunächst nur ein einzelnes Byte angeben ("UNUSED3") und dann erst das gepackte Feld ("UNUSED4").

Dies würde aber immer noch nicht funktionieren. Nach einem gepacked Feld beginnt der nächste Eintrag in dem Record wieder bei einem Byte mit gerade Stellenzahl. Wenn unser Feld nun 29 Elemente groß wäre, würde die nächste Information im 40. Byte stehen. Sie muß aber aus dem 39. kommen. Wir müssen das Feld nochmals kürzen, so daß es eine gerade Anzahl von Bytes einnimmt. Danach wird wieder ein einzelnes Byte angeben ("UNUSED5").

Nachdem wir diese Schwierigkeiten überstanden haben, folgt im 39. Byte eine Wert, wieviele Angaben in einem Sektor der Track/Sektor-Liste eines jedes Files stehen ("MAXPAIR"). Die Track/Sektor-Liste ist im DOS-Handbuch auf den Seiten 128/129 beschrieben. Auf sie soll hier nicht weiter eingegangen werden, da wir sie nicht benötigen, um den Catalog auszugeben.

Hierauf folgen 8 unbenutzte Bytes. Da sie bei einem Byte mit gerader Stellenzahl beginnen und eine eine gerade Anzahl haben, könne wir sie bedenkenlos in einem gepackten Feld zusammenfassen ("UNUSED6").

Nun kommen zwei Integerwerte. Sie stellen eine Maske dar, die das DOS benötigt, um Veränderungen an der Bit-Map (s.u.) vorzunehmen. Wir nennen sie "MASK", werden sie jedoch nicht mehr brauchen.

Dann folgen Informationen über den Aufbau der Diskette. Zunächst, wieviele Tracks sich auf der Diskette befinden ("TRKNO"), dann wieviel Sektoren sich in einem Track befinden ("SECNO") und schließlich, wieviele Byte in einem Sektor stehen ("BYTENO").

Nun folgt die schon genannte Liste, in der steht, welche Sektoren

belegt sind, und welche nicht. Sie wird "Bit-Map" genannt. Da ein Sektor nur entweder belegt oder frei sein kann, wird diese Informationen mit je einem Bit für jeden Sektor dargestellt. Hat das Bit den Wert 1, oder true, so ist der jeweilige Sektor frei, ansonsten belegt.

Normale DOS-Disketten haben 35 Tracks. Wir fassen diese Bit-Map als ein gepacktes Feld mit 35 Elementen von Typ "TRACKMAP" auf. "TRACKMAP" haben wir vorher definiert als ein gepacktes Boolean-Feld mit 32 Elementen. Da jeder Track normalerweise jedoch nur 16 Sektoren hat, bleiben die Bits 16 bis 31 unbenutzt und haben immer den Wert false.

Die Bytes 196 bis 255 im VTOC sind unbenutzt und heißen "UNUSED7".

Damit haben wir den Aufbau des VTOCs in einem gepackten Record zusammengefasst. Wir können ihn einlesen, und jede Information ansprechen.

"READINVTOC" liest das VTOC einer DOS-Diskette ein. Dazu benutzen wir den "UNITREAD"-Befehl, wobei wir aus dem Block 136 lesen. Im Grunde genommen handelt es sich nicht um einen Block, sondern um zwei DOS-Sektoren. Das VTOC befindet auf Track 17, Sektor 0. Wir können daraus den entsprechenden Block errechnet. Dieser ergibt sich aus 8 * Tracknummer, wozu der Wert hinzugerechnet wird, der sich aus der obigen Tabelle für den Sektor ergibt. 8 * 17 + 0 ergibt 136. Da wir wissen, daß der 0-te Sektor in der ersten Hälfte des errechneten Sektors liegt, brauchen wir keine Bytes zu verschieben, und können die Variable "VTOC" direkt einlesen.

"PRINTVTOC" gibt dann die VTOC-Informationen aus. Zunächst werden die Informationen ausgegeben, die wirklich von Wichtigkeit sind, wie Volume-Nummer, Anzahl der Track etc. Dann wird dargestellt, welche Sektoren belegt und welche frei sind. Belegte Sektoren werden mit einem Stern gekennzeichnet ("*").

Danach soll der Catalog ausgegeben. Sein Aufbau ist auf den Seiten 129-131 im DOS-Handbuch genau beschrieben.

Nun stellt sich uns ein schon bekanntes Problem. Wollen wir einen strukturierten Record zusammenstellen, der direkt eingelesen werden kann, wie das VTOC, so ist dies sehr problematisch. Dies hängt wieder damit zusammen, daß innerhalb eines gepackten Records alle gepackten Felder oder Records bei einem Byte beginnen, dessen Stellenzahl gerade ist. Nun nimmt ein File-Eintrag im Catalog eine ungerade Anzahl von Bytes ein. Aufgrund der oben beschriebenen Komplikationen wäre ein strukturierter Record für einen Catalog-Sektor äußerst aufwendig und zudem umständlich anzusprechen. Wir gehen einen anderen Weg.

In der Prozedur "READINCAT" wird ein Sektor des Catalogs eingelesen. Zur Errechnung der Blocknummer bedienen wir uns der Tabelle "SECLOC". Sie wurde vorher in der Prozedur "INIT" mit den Werten aus der obigen Tabelle gefüllt. Der Block wird zunächst in die Variable "BLOCK" eingelesen. Sie ist ein Feld aus 512 Bytes und dient als Zwischenspeicher. Befindet sich der Sektor in der rechten Hälfte eines Block, so werden in "BLOCK" 256 Bytes nach links verschoben, so daß das erste Element von "BLOCK" in jedem Fall das erste Byte des gewünschten Sektors enthält.

Nun wird der Inhalt des Catalogs aus der Variable "BLOCK" in "CAT"

übertragen. Dabei sprechen wir jedes Byte einfach durch einen Index an.

"CAT" ist ein strukturierter Record von Typ "CATSECTOR". Er ist entsprechend den Angaben des DOS-Handbuchs aufgebaut. Da wir die Informationen ja aus dem eingelesenen Sektor byteweise in diese Variable übertragen können wie die unbenutzten Bytes einfach auslassen.

Das erste Feld in "CAT" heißt "TRKLINK" und gibt an, auf welchem Track der nächste Sektor des Catalogs zu finden ist. "SECLINK" gibt an, in welchem Sektor. Haben beide Felder den Wert 0, so ist das Ende des Catalogs erreicht.

Nun folgen sieben File-Einträge. Jeder File-Eintrag ist vom Typ "FILEENTRY". Dieser ist wieder ein strukturierter Record.

Das erste Feld ("TSTRK") hier gibt an, auf welchem Track die Track/-Sektor-Liste für dieses File zu finden ist. Ist dieser Wert 0 oder 255, so ist das File gelöscht. Das zweite ("TSSEC") gibt die Sektornummer der Track/Sektor-Liste an. Nun folgt ein Wert, der angibt, um welchen Filetyp es sich handelt. Die Bedeutung dieses Wertes ist auf Seite 131 des DOS-Handbuchs erklärt. Die dort angegebenen Informationen sind allerdings unvollständig, da es noch die File-Typen "S" und "R" gibt. Sie sind beide unter DOS bekannt und haben die Werte 8 bzw. 16. Unser Programm wird auch sie bei der Ausgabe des Catalogs berücksichtigen.

Als nächstes kommt der Filename, der 30 Zeichen lang ist, und schließlich die Anzahl der Sektoren, die das File einnimmt.

Wenn wir nun den Record "CAT" mit den eingelesenen Werten gefüllt haben, können wir einen Catalog ausgeben. Dies über nimmt die Prozedur "PRINTCAT". Sie gibt die File-Einträge aus, wobei die Form eines "CATALOG"-Kommandos unter DOS übernommen wird.

Der Vorgang Catalog-Sektor einlesen und ausgeben wird solange wieder holt, bis wir am Ende des Catalogs angelangt sind. Dies dauert etwas länger als unter DOS.

Es erschließen sich nun einige Möglichkeiten. Wenn wir den DOS-Catalog lesen können, werden wir ihn auch verändern und zurückschreiben können. Wir könnten einen "DOS-Filer" schreiben, der dies vornimmt. Oder wir könnten Programme schreiben, die Daten von DOS-Disketten übernehmen. Oder man könnte sich einen Assembler schreiben, der ein Programm im DOS-Format erzeugt. Die Möglichkeiten sind unbegrenzt, zudem steht einem zur Programmierung das ausgereifte Pascal-System zur Verfügung.

Wichtige Prozeduren:

"INIT" : Initialisiert die Umrechnungstabelle DOS-Sektoren <--> POS-Blöcke

"READINVTOC" : Liest das VTOC einer DOS-Diskette in die Variable "VTOC" ein

"PRINTVTOC" : Gibt anhand von "VTOC" aus, welche Sektoren der DOS-Diskette belegt und welche frei sind

"CATALOG" : Gibt das Inhaltsverzeichnis der DOS-Diskette aus

"READINCAT" : Liest einen Sektor des Catalogs ein

"PRINTCAT" : Gibt einen Sektor des Catalogs, d.h. 7 Einträge aus

Programm "CATALOG.TEXT"

```
program doscat;

const
  dosunit = 5;

type
  byte = 0..255;
  { Bitmap fuer einen Track }
  trackmap = packed array [0..31] of boolean;
  { Record fuer VTOC einer DOS-Diskette }
  vtoc_type = packed record.
               unused1: byte;
               cattrk : byte;  { Track des ersten Catalog-Sektor }
               catsec : byte;  { Sektor des ersten Catalog-Sektor }
               dosvers: byte;  { DOS-Version }
               unused2: packed array [4..5] of byte;
               volume : byte;  { Volumenummer der Diskette }
               unused3: byte;
               unused4: packed array [8..37] of byte;
               unused5: byte;
               maxpair: byte;  { max. Anzahl der Angaben in einem Sektor }
               unused6: packed array [40..47] of byte; { einer T/S-Liste }
               mask   : packed array [0..1] of integer; { Maske fuer Bitmap }
               trkno  : byte;  { Anzahl der Tracks auf der Diskette }
               secno  : byte;  { Anzahl der Sektoren pro Track }
               byteno : integer; { Anzahl der Bytes in einem Sektor }
               bitmap : packed array [0..34] of trackmap; { Bitmaps fuer }
               unused7: packed array [196..255] of byte;  { 35 Tracks    }
             end;
  { Eintrag fuer ein File }
  file_entry = record
                tstrk : byte; { Track der T/S-Liste }
                tssec : byte; { Sektor der T/S-Liste }
                typ   : byte; { Filetyp }
                name  : string[30]; { Filename }
                seccnt: integer; { Anzahl der belegten Sektoren }
              end;
  { Record fuer einen Sektor des Catalogs }
  cat_sektor = record
                trklink: byte; { Zeiger auf naechsten Catalog Sektor (Track) }
                seclink: byte; { Zeiger auf naechsten Catalog Sektor (Sektor) }
                entries: packed array [1..7] of file_entry; { 7 Eintraege }
              end;                                           { in einem Sektor }
var vtoc:vtoc_type;
    cat:cat_sektor;
  secloc:array[0..15] of record
                          block:integer;
                          right:boolean;
                        end;

procedure init; { Initialisert Sektor->Block-Umrechnungstabelle }
begin
  secloc[ 0].block:=0; secloc[ 0].right:=false;
```

```
  secloc[ 1].block:=7; secloc[ 1].right:=false;
  secloc[ 2].block:=6; secloc[ 2].right:=true;
  secloc[ 3].block:=6; secloc[ 3].right:=false;
  secloc[ 4].block:=5; secloc[ 4].right:=true;
  secloc[ 5].block:=5; secloc[ 5].right:=false;
  secloc[ 6].block:=4; secloc[ 6].right:=true;
  secloc[ 7].block:=4; secloc[ 7].right:=false;
  secloc[ 8].block:=3; secloc[ 8].right:=true;
  secloc[ 9].block:=3; secloc[ 9].right:=false;
  secloc[10].block:=2; secloc[10].right:=true;
  secloc[11].block:=2; secloc[11].right:=false;
  secloc[12].block:=1; secloc[12].right:=true;
  secloc[13].block:=1; secloc[13].right:=false;
  secloc[14].block:=0; secloc[14].right:=true;
  secloc[15].block:=7; secloc[15].right:=true;
end;

procedure readin_vtoc; { Liest das VTOC von einer DOS-Diskette ein }
begin
  unitread(dosunit,vtoc,256,136);
end;

procedure print_vtoc;  { Gibt aus, welche Sektoren belegt, und welche }
var                    { frei sind }
  sector,track,index:integer;
begin
  write(chr(12));
  with vtoc do
    begin
      writeln('Catalog bei T',cattrk,',S',catsec,
              '  DOS Ver. ',dosvers,'  Vol ',volume);
      writeln(trkno,' Tracks  ',secno,' Sektoren  ',
              byteno,' Bytes/Sektor');
      writeln;
      writeln('                11111111112222222222333333');
      writeln('      012345678901234567890123456789O1234');
      writeln;
      for sector:= 0 to 15 do
        begin
          write('S',sector:2,' ');
          for track:= 0 to 34 do
            begin
              if sector > 7 then index:=sector-8
                            else index:=sector+8;
              if bitmap[track,index] then write(' ')
                                     else write('*');
            end;
          writeln;
        end;
    end;
end;

procedure catalog; { Gibt den Catalog einer DOS-Diskette aus }
var t,s:integer;
linecnt:integer;

  procedure readin_cat(track,sector:integer); { Liest einen Sektor ein, und }
  var block : packed array [0..511] of byte;  { uebertraegt ihn in den      }
    entr,cnt:integer;                         { Record 'cat' }
  begin
    { Sektor einlesen }
```

```
  unitread(dosunit,block,512,136+secloc[sector].block);
  { falls der Sektor sich im zweiten Teil des Blocks befindet, }
  { um 256 Bytes nach links verschieben }
  if secloc[sector].right then
    moveleft(block[256],block[0],256);
  { Daten interpretieren }

  with cat do
    begin
      trklink:=block[1];
      seclink:=block[2];
      for entr:=1 to 7 do
        with entries[entr] do
          begin
            tstrk:=block[11+(entr-1)*35+0];
            tssec:=block[11+(entr-1)*35+1];
            typ  :=block[11+(entr-1)*35+2];
            name:='                              ';
            for cnt:=1 to 30 do
              name[cnt]:=chr(block[11+(entr-1)*35+2+cnt]);
            seccnt:=block[11+(entr-1)*35+33];
          end;
    end;
end;

procedure print_cat; { Fileeintraege ausgeben }
var entr:integer;
begin
  with cat do
    begin
      for entr:=1 to 7 do
        with entries[entr] do
          if not (tstrk in [0,255]) then
            begin
              if linecnt=20 then
                begin
                  writeln;
                  write('Bitte <ret> druecken');
                  readln;
                  linecnt:=0;
                  write(chr(12));
                end;
              if typ>127 then begin
                                write('*');
                                typ:=typ-128;
                              end
                         else write(' ');
              case typ of
                 0: write('T ');
                 1: write('I ');
                 2: write('A ');
                 4: write('B ');
                 8: write('S ');
                16: write('R ')
              end;
              if not (typ in [0,1,2,4,8,16])
                then write('? ');
              write(seccnt:3);
              writeln(' ',name);
              linecnt:=linecnt+1;
            end;
```

```
      end;
    end;

 begin
   t:=vtoc.cattrk;
   s:=vtoc.catsec;
   linecnt:=0;
   { Catalog-Sektoren bis zum Ende einlesen und ausgeben }
   repeat
     readin_cat(t,s);
     print_cat;
     t:=cat.trklink;
     s:=cat.seclink;
   until (s=0) and (t=0);
 end;

 begin
   init;
   write(chr(12),'Bitte DOS-Diskette in Laufwerk ',dosunit,' einlegen. <ret>');
   readln;
   readin_vtoc;
   print_vtoc;
   writeln;
   write('Bitte <ret> druecken');
   readln;
   write(chr(12));
   catalog;
   writeln;
   write('Bitte DOS-Diskette entfernen <ret>');
   readln;
 end.
```

2.6 POS 1.1 und DOS 3.3 auf einer Diskette

Wir wissen inzwischen, wie man von Pascal aus das POS-Directory und auch den DOS-Catalog anspricht. In diesem Kapitel werden wir dieses Wissen anwenden, um ein besonderes Formatier-Programm zu schreiben. Das Ziel ist es, auf einer einzigen Diskette sowohl DOS- als auch POS-Dateien zu haben.

Dazu müssen wir die Diskette in zwei Bereiche aufteilen, je einen für jedes Betriebssystem. Pascal erwartet das Directory auf Track 0 und DOS den Catalog auf Track 17. Diese Stellen sind verbindlich. Nun ergeben sich daraus aber keine Probleme. Wir teilen unsere Diskette so auf, das die ersten 16 Tracks von POS und die Tracks 17 bis 35 von DOS benutzt werden. Diese Aufteilung findet sich in Bild 2.6.1.

Track(s)	belegt mit
0	POS-Directory
1-16	POS-Dateien
17	DOS-Catalog
17-35	DOS-Dateien

Bild 2.6.1: Die Aufteilung einer Diskette in DOS und POS

Es ergibt sich darurch nur eine Einschränkung, nämlich, daß die Diskette nur unter Pascal gebootet werden kann. Um DOS zu booten werden die Tracks 0 bis 2 benötigt. Diese sind hier aber von Pascal belegt. Das Booten unter POS bereitet keine Probleme.

Wie erreichen wir aber, daß DOS und POS nicht auf die Diskettenbereiche zugreifen, die für das jeweils andere Betriebssystem reserviert sind ? Wir müssen das Directory und den Catalog darauf einrichten.

Für POS sind die Tracks 0 bis 16 reserviert. 17 Tracks ergeben 17*8= 136 Blöcke. Da in Directory in Feld "BLOCKNO" angegeben ist, auf wieviele aufeinanderfolgende Blöcke vorhanden sind, geben wir hier einfach den Wert 135 für den letzten Block an. Für POS hat die Diskette nun nur 16 Tracks. Die Tracks 17 bis 35 existieren also theoretisch nicht, und werden auch nicht angesprochen. Damit ist der DOS-Teil der Diskette geschützt.

Unter DOS stehen die Informationen über die Diskette selber im VTOC. Es gibt hier zwar auch die Angabe "TRKNO", die angibt, wieviele Tracks auf der Diskette vorhanden sind. Wir können diese Angabe jedoch nicht für unsere Zwecke benutzen. Der Grund ist, daß an dieser Stelle steht, wieviele Tracks, beginnend bei Track 0 es auf der Diskette gibt. Wenn wir hier 18 (35 vorhandene Tracks minus 17 reservierte) hineinschreiben würden, hätte DOS Zugriff auf die Tracks 0 bis 18. Genau das wollen wir aber verhindern, "TRKNO" hilft uns also nicht weiter.

Anstatt dessen benutzen wir einfach die "BITMAP", die angibt, welche Sektoren belegt sind, und welche nicht. Wenn wir alle Sektoren, die für POS reserviert sind einfach als "belegt" kennzeichnen, wird DOS nicht auf sie zugreifen, und der POS-Teil der Diskette ist geschützt

Bei unserem Programm gehen wir davon aus, daß die zu zu bearbeitende Diskette schon formatiert ist. Dies sollte zweckmäßigerweise unter POS geschehen. Ein unter DOS formaterte Diskette unterscheidet sich von einer Pascal-Diskette auch darin, daß die physikalische Anordnung der Sektoren auf der Diskette anders ist. Für den logischen Zugriff hat dies keine Folgen, jedoch ist diese Anordnung unter DOS nicht optimal. Wenn wir eine unter DOS formatierte Diskette als Pascal-Diskette verwenden, wird der Zugriff auf die Diskette wesentlich langsamer.

Unser Formatier-Programm formatiert im Grunde genommen die Diskette garnicht, es schreibt nur die nötigen Directory- und Cataloginformationen. Deshalb gliedert es sich auch in zwei Teile.

Der erste Teil, "FORMATDOS" schreibt das VTOC und den Catalog auf die Diskette. Den Aufbau des VTOC übernehmen wir aus Kapitel 2.5. Das VTOC wird zunächst in der Prozedur "SETUPVTOC" mit den gewünschten Informationen gefüllt. Bis auf "BITMAP" werden alle Felder so gesetzt, wie sie auf einer normalen DOS-Diskette vorkommen. Hier darf man sich allerdings nicht in allen Fällen von den Angaben auf Seite 132 des DOS-Handbuchs leiten lassen. Die hier angegebenen Werte sind teilweise falsch. So muß z.B. im Byte 35 ein 10 (hex) anstatt eines 0F(hex) stehen.

Im Feld "BITMAP" werden, wie oben beschrieben, die Tracks 0 bis 16 als belegt vermerkt. Track 17 muß auch noch belegt werden, da hier ja der Catalog steht. Danach wird mit der Prozedure "WRITEVTOC" das VTOC auf die Diskette gebracht.

Das Schreiben des Catalogs ist relativ einfach. Wenn man sich eine neu formatierte DOS-Diskette z.B. mit einem sog. Sektor-Editor anschaut, stellt man fest, daß im Catalog bis auf die Bytes 1 und 2 nur der Wert 0 vorkommt. Diese zwei Bytes geben an, auf welchem Sektor die Fortsetzung des Catalogs zu finden ist. Die Prozedur "WRITECAT" setzt diese Bytes entsprechend und schreibt insgesamt 15 Sektoren auf die Diskette auf dem Track 17. Wie aus Kapitel 2.5 bekannt sind die Sektoren etwas anders angeordnet als die POS-Blöcke. Um hier abzuhelfen benuzten wir wieder die Tabelle "SECLOC".

Mit der Prozedur "FORMATPOS" wird das Directory auf die Diskette geschrieben. Das Aussehen des Directorys übernehmen wir aus Kapitel 2.2. In der Prozedur "SETUPDIR" wird es mit den nötigen Informationen gefüllt. Als Diskettenname geben wir "BLANK" an. Weiterhin wird wie oben beschrieben, die Anzahl der Blöcke auf 136 begrenzt. Das Datum lesen wir aus dem Speicher aus. Die Vorgehensweise hierfür ist in Kapitel 3.4 beschrieben. Schließlich wird das Directory mit der Prozedur "WRITEDIR" auf die Diskette geschrieben.

Damit ist das Programm abgeschlossen. Man kann nun die Diskette unter DOS und POS verwenden. Dabei hat man fast gleichviel Platz, für POS 68 KByte und für DOS 72 KByte.

Der Leser kann natürlich den Raum für POS weiter einschränken, indem im Directory weniger Blöcke vermerkt und im VTOC die entsprechenden Sektoren freigesetzt werden.

Wichtige Prozeduren

"GETUNITNO" : Fragt den Benutzer, in welchem Laufwerk die zu bearbeitende Diskette liegt.

"FORMATDOS" : Schreibt die nötigen DOS-Informationen auf die Diskette

"FORMATPOS" : Schreibt die nötigen POS-Informationen auf die Diskette

Programm "DOSPOS.TEXT"

```
{$S+}
programm dospos;

type

  { POS-Directory Definitionen }

  vname=string[7]; { Ein Volumename }
  fname=string[15]; { Ein Filename }
  daterec = packed record { das Datum }
              month: 0..12;
              day:   0..31;
              year:  0..100;
            end;
  fkind = (vol,bad,code,text,info,data,graf,foto,secr); { moegliche Filetypen }
  direntry = record { ein Eintrag im Directory }
               firstblock: integer; { Block, bei dem das File anfaengt }
               lastblock: integer;  { Block, bei dem das File endet }
               case filekind: fkind of
                    { nur beim Eintrag #0 }
```

```
                    vol,secr: (diskvol: vname;    { Name der Diskette }
                               blockno: integer; { Anzahl der Bloecke }
                               fileno:integer;   { Anzahl der Files }
                               accesstime: integer; { unbenutzt }
                               lastboot: daterec);  { Datum, an dem die     }
                                                    { zuletzt benutzt wurde }
                    { Normale File-Eintraege }
                    bad,code,text,
                    info,data,graf,
                    foto: (filename: fname; { Filename }
                           endbytes: 1..512; { Bytes im letzten Block }
                           lastaccess: daterec)  { Datu, an dem das File }
                end;
  direc = array[0..77] of direntry; { Directory mit 78 Eintraegen }

  { DOS-VTOC Definition }

  byte = 0..255;
  trackmap = packed array [0..31] of boolean;
  vtoc_type = packed record
                 unused1: byte;
                 cattrk : byte;
                 catsec : byte;
                 dosvers: byte;
                 unused2: packed array [4..5] of byte;
                 volume : byte;
                 unused3: byte;
                 unused4: packed array [8..37] of byte;
                 unused5: byte;
                 maxpair: byte;
                 unused6: packed array [40..47] of byte;
                 mask   : packed array [0..1] of integer;
                 trkno  : byte;
                 secno  : byte;
                 byteno : integer;
                 bitmap : packed array [0..34] of trackmap;
                 unused7: packed array [196..255] of byte;
               end;

var unitno:integer;

procedure get_unitno; { Fragt Benutzer, auf welchem Laufwerk }
var ch:char;          { formatiert werden soll               }
begin
  repeat
    writeln(chr(12),'DOS-POS FORMATTER PROGRAM');
    write('FORMAT WHICH DISK (4,5) ? ');
    read(ch);
  until ch in ['4','5'];
  unitno:=ord(ch)-48;
  writeln;
  writeln('NOW FORMATTING DISK IN DRIVE #',unitno);
end;

procedure formatdos; { Formatiert den DOS-Teil der Diskette }
var secloc:array[0..15] of record
                           block:integer;
                           right:boolean;
                          end;
    vtoc:vtoc_type;
```

```
procedure init;  { Initialisert Block-Sektor Umrechnungstabelle }
begin
  secloc[ 0].block:=0; secloc[ 0].right:=false;
  secloc[ 1].block:=7; secloc[ 1].right:=false;
  secloc[ 2].block:=6; secloc[ 2].right:=true;
  secloc[ 3].block:=6; secloc[ 3].right:=false;
  secloc[ 4].block:=5; secloc[ 4].right:=true;
  secloc[ 5].block:=5; secloc[ 5].right:=false;
  secloc[ 6].block:=4; secloc[ 6].right:=true;
  secloc[ 7].block:=4; secloc[ 7].right:=false;
  secloc[ 8].block:=3; secloc[ 8].right:=true;
  secloc[ 9].block:=3; secloc[ 9].right:=false;
  secloc[10].block:=2; secloc[10].right:=true;
  secloc[11].block:=2; secloc[11].right:=false;
  secloc[12].block:=1; secloc[12].right:=true;
  secloc[13].block:=1; secloc[13].right:=false;
  secloc[14].block:=0; secloc[14].right:=true;
  secloc[15].block:=7; secloc[15].right:=true;
end;

procedure setup_vtoc; { Fuellt VTOC mit den gewuenschten Informationen }
var cnt,s:integer;
begin
  with vtoc do begin
    unused1:=0;
    cattrk:=17;  { Catalog auf Track 17 }
    catsec:=15;  {              Sektor 15 }
    dosvers:=3;  { Dosversion }
    fillchar(unused2,sizeof(unused2),chr(0));
    volume:=254; { Volume }
    unused3:=0;
    fillchar(unused4,sizeof(unused4),chr(0));
    unused5:=0;
    maxpair:=122; { max. Anzahl der Angaben in einer T/S Liste }
    fillchar(unused6,sizeof(unused6),chr(0));
    mask[0]:=-1;  { alle 16 Bits auf 1 setzen }
    mask[1]:=0;   { alle 16 Bits auf 0 setzen }
    trkno:=35;    { 35 Tracks mit je }
    secno:=16;    { 16 Sektoren mit je }
    byteno:=256; { 256 Bytes }
    fillchar(bitmap,sizeof(bitmap),chr(0)); { Alle Sektoren belegen }
    for cnt:=18 to 34 do    { Tracks 18 bis 34 frei }
      for s:=0 to 15 do
        bitmap[cnt,s]:=true;
    fillchar(unused7,sizeof(unused7),chr(0));
  end;
end;

procedure write_vtoc; { Schreibt VTOC auf die Diskette }
begin
  unitwrite(unitno,vtoc,256,136);
end;

procedure write_cat;  { Schreibt einen leeren Catalog auf die Diskette }
var block : packed array [0..511] of byte;
    offset,t,sector:integer;
begin
  t:=17;
  for sector:= 15 downto 1 do
    begin
      { Erst ganzen Block einlesen, da nur eine Haelfte veraendert wird }
```

```
        unitread(unitno,block,512,136+secloc[sector].block);
        { linke oder rechte Haelfte veraendern ? }
        if secloc[sector].right
          then offset:=256
          else offset:=0;
        { ganzen Sektorinhalt loeschen }
        fillchar(block[offset],256,chr(0));
        { Zeiger auf naechsten Catalog-Sektor setzen }
        if sector>1 then begin
                           block[1+offset]:=t;         { Links setzen }
                           block[2+offset]:=sector-1;
                         end
                    else begin { Ende des Catalogs }
                           block[1+offset]:=0;
                           block[2+offset]:=0;
                         end;
        { Block zurueckschreiben }
        unitwrite(unitno,block,512,136+secloc[sector].block);
      end;
  end;


begin
  init;
  setup_vtoc;
  write_vtoc;
  write_cat;
end;

procedure formatpos; { Formatiert den POS-Teil der Diskette }
var directory:direc;

  procedure setup_dir; { Directory mit den gewuenschten Informationen fuellen }
  var sprchdatum : record case boolean of
                     true : (datum:^daterec);
                     false: (adresse:integer);
                   end;
  begin
    with directory[0] do begin { Nur Eintrag #0 setzen }
      firstblock:=0;     { Directory-Beginn }
      lastblock:=5;      { Directory-Ende   }
      filekind:=vol;     { Filetyp: vol     }
      diskvol:='BLANK'; { BLANK als Name   }
      blockno:=135;      { Nur 135 Bloecke! }
      fileno:=0;         { Noch keine Files }
      accesstime:=0;
      sprchdatum.adresse:= -21992; { Siehe Liste in Kap. 3.2 }
      lastboot:=sprchdatum.datum^; { Datum wie im Speicher   }
    end;
  end;

  procedure write_dir; { schreibt das Directory auf die Diskette }
  begin
    unitwrite(unitno,directory,sizeof(directory),2);
  end;

begin
  setup_dir;
  write_dir;
end;
```

```
begin
  get_unitno; { 'unitno' einlesen  }
  format_dos; { DOS-Teil formatieren }
  format_pos; { POS-Teil formatieren }
  write('PUT SYSTEM DISK IN DRIVE #4 (<RET>)');
  readln;
end.
```

2.7 Foto, ein neuer Filetyp

Wie wir in Kapitel 2.2 gesehen haben, stellt POS eine ganze Reihe von Filetypen zur Verfügung. In Apple-Pascal werden normalerweise aber nur die Typen Text, Code und Data verwendet.

Mit den zwei Prozeduren "PIXSAVE" und "PIXLOAD" nutzen wir einen der bisher nicht genutzten Filetypen aus, nämlich den Foto-Typ. Gleichzeitig ermöglichen wir es, eine Graphikseite auf Diskette zu speichern. Dies war bishar nicht möglich.

Dazu ist natürlich ein Trick notwendig. Er besteht darin, den Inhalt der Graphikseite zu einer Variablen zu machen. Dabei fassen wir eine 8 KByte große Seite als PACKED ARRAY mit 8191 Bytes auf. Dann ist es noch nötig, unsere Variable bei Speicherstelle 8192 (hexadezimal $2000) beginnen zu lassen. Dies wird mittels eines Zeigers erreicht. Die genaue Arbeitsweise ist in Kapitel 3.1 beschrieben.

Bis jetzt können wir eine Graphikseite unter einem bestimmten Namen auf die Diskette schreiben. Wie weisen wir diesem File den Typ FOTO zu ?

Dies ist ziemlich einfach. POS setzt nämlich die Filetypen in Abhängigkeit vom Filenamen. Endet er mit ".FOTO", so ist es ein Fotofile.

Der Leser kann dies ausprobieren, indem er in den Filer geht und mit dem M)ake-Kommando einige Files "macht". Dabei sollten Namen benutzt werden, die mit ".FOTO", ".INFO" oder ".GRAF" enden. Wird dann mit E das Directory aufgelistet, dann erscheinen in der letzten Spalte die ungewohnten, bis jetzt ungenutzten Filetypen.

An unsere zwei Prozeduren wird also eine Filename übergeben, an den die Endung ".FOTO" angehängt wird.

Wichtige Variablen und Typen

"BILD" : 8191 Bytes großes Feld (Entspricht der Größe einer Hires-Seite)

"BILDVAR" : Record, dessen Adresse im Speicher mittels eines Zeigers festgelegt werden kann. Entspricht der Struktur in Kapitel 3.1, nur daß hier nicht 1, sondern 8191 Bytes als Daten genommen werden.

"PIX" : File für ein Hiresbild

Wichtige Prozeduren

"PIXSAVE" : Hängt ".FOTO" an den Filenamen an und legt die Variable "X" von Typ "BILDVAR" über die Hires-Seite. Diese wird dann mit "PUT" in

das File "PIX" geschrieben.

"PIXLOAD" : Hängt ".FOTO" an den Filenamen an und liest mit "GET" ein Hires-Bild aus dem File "PIX". Legt dann die Variable "X" über die Hires-Seite und holt den Inhalt des Bildes aus dem Filepuffer.

Progamm "FOTO.TEXT"

```
procedure pixsave (filename:string); { schreibt ein Bild aus }
                                     { dem Hiresspeicher auf die Diskette }
type bild = packed array[0..8191] of 0..255;
  bildvar = record case boolean of
              true: (adresse:integer);
              false:(zeiger:^bild)
            end;

var pix: file of bild;
      x: bildvar;

begin
  filename:=concat(filename,'.FOTO');
  x.adresse:=8192;
  pix^:=x.zeiger^;
  rewrite(pix,filename);
  put(pix);
  close(pix,lock);
end;

procedure pixload (filename:string); { laedt ein Bild von der }
                                     { Diskette in den Hiresspeicher }
type bild = packed array[0..8191] of 0..255;
  bildvar = record case boolean of
              true: (adresse:integer);
              false:(zeiger:^bild)
            end;

var pix: file of bild;
      x: bildvar;

begin
  filename:=concat(filename,'.FOTO');
  reset(pix,filename);
  get(pix);
  close(pix,normal);
  x.adresse:=8192;
  x.zeiger^:=pix^;
end;
```

Wer mit den zwei Prozeduren arbeitet wird zweierlei feststellen. Zunächst geht das Laden eines Bildes entschieden schneller als unter DOS 3.3. Dies hängt damit zusammen, daß POS allgemein schneller als DOS ist, nicht nur beim Laden von Fotos. Die zweite Eigenart der Prozeduren ist ein Nachteil. Beide verbrauchen nämlich über 8 KByte Speicherplatz, während sie aufgerufen sind. Ist durch das Hauptprogramm schon viel Speicher verbraucht, so kann es zu einem "STACK OVERFLOW"-Error kommen, der Datenverlust bedeutet. Man kann hier abhelfen, indem man das Programm segmentiert.

2.8 Textfiles und Codefiles

In diesem Kapitel werden wir näher auf den Aufbau von Text- und Codefiles eingehen. Bei Textfiles wird dargestellt, wie die Editor-Informationen zu Beginn eines jeden Textfiles aussehen und bei den Codefiles werden wir uns das sogenannte "Segment Dictionary" anschauen.

Das Format von Textfiles ist im "Apple Operating System" Handbuch auf der Seite 266 beschrieben. Die dort angegebenen Informationen sind jedoch unvollständig. Es wird nur gezeigt, welche Bedeutung die sogenannten "DLE"'s haben. Auf sie gehen wir in Kapitel 5.2 näher ein. Über den Aufbau der Editorinformationen ist jedoch nichts vorhanden.

Bei jedem Textfile sind die ersten zwei Blöcke reserviert für Informationen, die der Editor benötigt. Erst im dritten Block beginnt der eigentliche Text. Daher nimmt ein leeres Textfile auch mehr als einen Block auf der Diskette ein. Die Informationen für den Editor sind im wesentlichen die, die man im Editor mit dem "SE"-Kommando verändern kann. Z.B. rechter oder linker Rand sind solche Werte. Wird ein Textfile z.B. mit "RESET" angesprochen, so werden die ersten zwei Blöcke automatisch überlesen.

Wie fast alles in POS, lassen sich auch diese Editor-Informationen mit einem Record darstellen. Er ist in Bild 2.8.1 dargestellt.

```
TEXTINFO: PACKED RECORD
   UNUSED1          : PACKED ARRAY[0..3] OF CHAR;
   MARKERNAME       : PACKED ARRAY[0..9,0..7] OF CHAR;
   UNUSED2          : PACKED ARRAY[0..9] OF CHAR;
   MARKERADRESSE    : PACKED ARRAY[0..9] OF INTEGER;
   AUTOINDENT       : INTEGER;
   FILLING          : INTEGER;
   TOKENDEF         : INTEGER;
   LEFTMARGIN       : INTEGER;
   RIGHTMARGIN      : INTEGER;
   PARAMARGIN       : INTEGER;
   COMMANDCH        : CHAR;
   DATECREATED      : PACKED RECORD
                        MONAT : 0..12;
                        TAG   : 0..31;
                        JAHR  : 0..99;
                      END;
   LASTUSED         : PACKED RECORD
                        MONAT : 0..12;
                        TAG   : 0..31;
                        JAHR  : 0..99;
                      END;
   UNUSED3          : PACKED ARRAY[0..379] OF CHAR;
END;
```

Bild 2.8.1: Ein strukturierter Record für die Editorinformationen

Wir gehen nun jedes einzelne Feld durch, und erklären seine Bedeutung.

Das erste Feld "UNUSED1" besteht aus 10 Bytes, und ist unbenutzt.

Darauf folgt eine Liste der Namen der Marken ("MARKENNAME"). Es gibt maximal 10 Marken mit je 8 Zeichen. Sie werden hier als ein Feld von einzelnen Zeichen dargestellt. Ist das erste Zeichen eines Markennamens ein CHR(0), so ist diese Marke nicht gesetzt. Dann kommen wieder 10 unbenutzte Bytes ("UNUSED2").

Das Feld "MARKERADRESSE" enthält die Positionen der 10 Marken. Diese sind Integerwerte und geben an, auf welches Zeichen, relativ zum Textbeginn, eine Marke zeigt.

Nun folgen die Informationen, die mit dem "SE"-Kommando angezeigt und verändert werden können. "AUTOINDENT","FILLING" und "TOKENDEF" geben an, ob die jeweilige Option gesetzt ist oder nicht. "LEFTMARGIN","RIGHTMARGIN" und "PARAMARGIN" erhalten die Werte für den linken und rechten Rand, und wieweit bei einem neuen Absatz eingerückt werden soll. "COMMANDCH" enthält ein reserviertes Zeichen. Wenn das "M)argin"-Kommando eingegeben wird, und der Editor trifft auf dieses Zeichen am Beginn einer Zeile, so wird diese Zeile nicht neu formatiert. Dies ist nützlich, wenn man ein Formatierprogramm hat, in dem die Formatkommandos im Text enthalten sein müssen. Diese Kommandos fangen oft z.B. mit einem Punkt an, und gehören nicht zum eigentlichen Text.

Die nächsten zwei Einträge ("DATECREATED" und "LASTUSED") geben an, wann der Text zum erstem Mal, und wann er zuletzt editiert wurde. Beide enthalten ein Datum, das als ein gepackter Record aus Tag, Monat und Jahr dargestellt wird.

Schließlich folgen noch 380 unbenutzte Bytes ("UNUSED3").

Insgesamt ist dieser Record 512 Bytes groß. Er würde auf der Diskette einen Block einnehmen. Jedoch sind bei Textfiles zwei Blöcke reserviert. Der zweite Block ist unter Apple Pascal 1.1 unbenutzt. Er wird von dem Standardeditor nicht angeprochen und ist mit dem Wert 00 gefüllt. Es gibt jedoch Editoren, die diesen zweiten Block für weitere Informationen nutzen. Auch könnte es eines Tages einen neuen Standardeditor geben, der auch beide Blöcke benutzt. Dann wären alle alten Textfiles schon darauf vorbereitet. Im Moment jedoch geht mit jedem Textfile ein Block Diskettenplatz unbenutzt verloren.

Man kann diesen Record direkt mit der "BLOCKREAD" Prozedur aus dem ersten Block eines Textfiles einlesen. Dies ist in Kapitel 5.1 demonstriert.

Der Aufbau von Codefiles ist im "Apple Pascal Operating System" auf den Seiten 266 bis 270 genau beschrieben. Wir wollen ihn hier nochmals durchgehen. Es schließt sich ein Programm an, das die Informationen über ein Codefile einliest und auf dem Bildschirm ausgibt.

Block 0 (d.h. der erste Block) eines jeden Codefiles enthält das sogenannte "segment dictionary". Es enthält Informationen über jedes Segment des Codefiles. Hier ist anzumerken, daß im Grunde genommen jedes Codefile segmentiert ist. Bei einem normalen Programm, das nicht die "SEGMENT"-Anweisung benutzt, wird nur das erste Segment benutzt. Andere Programme können bis zu 15 weitere Segmente benutzen

Das "segment dictionary" ist ein strukturierter Record. Die ersten fünf Einträge sind Felder mit 16 Elementen, je eins für jedes Segment.

Der erste Eintrag "DISKINFO" gibt mit den Einträgen "CODELENG" und "CODEADDR" an, wie lang der Code für ein Segment ist, und wo im File er beginnt. Dann folgen die Namen der Segmente ("SEGNAME"). Jeder Name kann 7 Zeichen lang sein. Der Name des ersten Segments entspricht immer dem Namen, der hinter der Anweisung "PROGRAM" am Anfang eines jedes Programmtextes steht.

Im Eintrag "SEGKIND" folgt eine Angabe darüber, um was für ein Segment es sich handelt. Hier gibt es 8 Möglichkeiten.

Bei "LINKED" handelt es sich um einen Code, der so ausführbar ist. Eventuelle Units oder Externals sind schon eingebunden, oder es kommen keine vor.

"HOSTSEG" ist Code, der Externals aufruft. Diese sind noch nicht eingebunden, wodurch der Code noch nicht ausführbar ist. "SEGPROC" wird auf dem Apple nicht benutzt.

Bei einem "UNITSEG" handelt es sich um eine Unit. Sie ist nicht ausführbar, und muß erst noch in ein Programm eingebunden werden.

Ein "SEPRTSEG"-Code ist einzeln compiliert worden. Das ist unter Apple-Pascal nur der Fall bei Assemblerunterprogrammen. Der Code, den der Assembler erzeugt, hat diesen Typ.

"UNLINKEDINTRINS" bezeichnet Code für eine Intrinsic Unit. Es wurden allerdings notwendige Units oder Externals noch nicht eingebunden. Dagegen ist "LINKEDINTRINSIC" eine komplette Intrinsic-Unit, die fertig ist. Dies trifft bei allen Units der "SYSTEM.LIBRARY" zu.

Der Typ "DATASEG" wird zusammen mit einer Intrinsic-Unit verwendet. Er gibt an, wieviele Bytes die entsprechende Intrinsic-Unit für Variablen benötigt. Diese werden daraufhin reserviert. Z.B. die "Turtlegraphics" Unit hat ein solches Datensegment.

Der nächste Eintrag im "segment dictionary" heißt "TEXTADDR". Wenn ein Programm, das eine normale oder eine Intrinsic-Unit benutzt, compiliert wird, benötigt der Compiler Informationen über die in der Unit enthaltenen Prozeduren und Funktionen. Diese werden dann in der Unit im Textformat bereitgehalten. "TEXTADDR" gibt an, an welcher Stelle innerhalb des Unit-Codes sie zu finden sind. Handelt es sich nicht um eine Unit, so steht hier immer der Wert Null.

Als nächstes kommen einige weitere Informationen über ein Segment ("SEGINFO"). In "SEGNUM" steht, unter welcher Segmentnummer das jeweilige Segment intern laufen soll. Diese Nummer kann von 0 bis 255 gehen und hat mit der Nummer des Segments innerhalb des Codefiles nichts zu tun.

"MTYPE" gibt an, um welchen Code es sich handelt. Das UCSD-Pascal-System gibt es auf verschiedenen Computern mit verschiedenen Prozessoren. Wenn z.B. in diesem Feld der Wert 8 steht, so handelt es sich um eine Assemblerroutine für den 6800-Prozessor. Der Apple-6502 hat den Wert 7. Der normale P-Code auf dem Apple erhält den Wert 2. Bei Apple-Files kommt auch noch der Wert 0 vor, der angibt, daß es sich um Code handelt, der nicht identifiziert werden konnte.

Der Eintrag "VERSION" gibt an, unter welcher Pascalversion der Code erzeugt wurde. Hier steht für Apple-Pascal 1.1 der Wert 2. Der Vorgänger, die Version 1.0 hat der Wert 1.

Das hier abgedruckte Programm liest für ein bestimmtes Code-File diese Informationen ein, und gibt sie auf dem Bildschirm aus. Es ist interessant, mit dem Programm etwas zu "spielen", und sich z.B. den "SYSTEM.EDITOR" anzuschauen. Manchmal erlebt man auch Überraschungen. Es kommt ab und zu vor, daß in dem Feld "SEGNAME" ein Copyright-Vermerk untergebracht ist.

Wie aus dem Handbuch ersichtlich, enthalten Units noch mehr Informationen. Wir werden hier nicht weiter darauf eingehen, zumal es das Programm "LIBMAP" auf der Diskette "APPLE3:" gibt. Damit hat man Zugriff auf diese weiteren Informationen. Hier noch ein Tip: man kann "LIBMAP" auch auf normale Codefiles anwenden, man ist nicht auf Units beschränkt.

Programm "CODEFILE.TEXT"

```
program codefile;

type segment_directory = record
                            diskinfo: array[0..15] of
                                        record
                                          codeleng,codeaddr:integer;
                                        end;
                            segname : array[0..15] of
                                        packed array[0..7] of char;
                            segkind : array[0..15] of
                                        (linked,hostseg,segproc,unitseg,
                                         seprtseg,unlinked_intrins,
                                         linked_intrins,dataseg);
                            textaddr: array[0..15] of integer;
                            seginfo : packed array[0..15] of
                                          packed record
                                            segnum : 0..255;
                                            mtype  : 0..15;
                                            unused : 0..1;
                                            version: 0..7;
                                          end;
                            intrins_segs: set of 0..31;
                          end;

var info:segment_directory;
    codef:file;
    fname:string;

procedure hole_name;
begin
  write('Name des Codefiles >');
  readln(fname);
end;

procedure lese_info;
var dummy:integer;
begin
  reset(codef,fname);
  dummy:=blockread(codef,info,1);
  close(codef);
end;

procedure gebe_info_aus;
var i,j:integer;
```

```
begin
  writeln(chr(12));
  writeln('+--+-----+-----+--------+----------+-----+---+----+----+');
  writeln('! #!codel!codea!  name  !   kind   !taddr!num!mtyp!vers!');
  writeln('+--+-----+-----+--------+----------+-----+---+----+----+');
  for i:= 0 to 15 do
    with info do begin
      write('!',i:2,'!');
      write(diskinfo[i].codeleng:5,'!',diskinfo[i].codeaddr:5,'!');
      for j:=0 to 7 do
        write(segname[i,j]);
      write('!');
      case segkind[i] of
        linked            : write('  linked  ');
        hostseg           : write(' host seg ');
        segproc           : write(' seg proc ');
        unitseg           : write(' unit seg ');
        seprtseg          : write(' seprtseg ');
        unlinked_intrins: write('unlnkd int');
          linked_intrins: write(' lnkd int ');
        dataseg           : write(' data seg ')
      end;
      write('!');
      write(textaddr[i]:5,'!');
      with seginfo[i] do
        begin
          write(segnum:3,'!');
          write(mtype:4,'!');
          writeln(version:4,'!');
        end;
    end;
  writeln('+--+-----+-----+--------+----------+-----+---+----+----+');
end;

begin
  hole_name;
  lese_info;
  gebe_info_aus;
end.
```

2.9 Wenn READLN zu langsam ist

Wer unter Pascal einen Text aus einem File einlesen will, der öffnet das betreffende File und liest dann Strings mit der Prozedure "READLN" ein. Dies funktioniert ohne Zweifel. Jedoch ist die "READLN"-Prozedur äußerst langsam. Zum Einlesen von 100 Strings, die 40 Zeichen lang sind, benötigt das System knapp 50 Sekunden. Dies ist ziemlich lang und in diesem Kapitel wird beschrieben, wie man die Einlesezeit um über 90 Prozent verbessern kann.

Daß unsere Lösung allerdings nur auf das Einlesen eines Textes mit bekannter Länge beschränkt ist, wird am Ende des Kapitels näher beschrieben.

Da wir die Prozedur "READLN" natürlich nicht direkt ändern können, müssen wir sie von Grund auf neu schreiben. Wir gehen dabei jedoch einen anderen Weg, als die Standardprozedur. Diese benutzt nämlich den "GET"-Befehl, was auch für die Langsamkeit der Prozedur verantwortlich ist.

Zunächst muß uns der Aufbau eines Textfiles klar sein. Wie in Kapitel 2.8 beschrieben, beinhalten die ersten zwei Blöcke eines Textfiles Informationen für den Editor. Danach beginnt der eigentliche Text. Dessen Aufbau ist im "Apple Pascal Operating System" Handbuch auf der Seite 266 kurz dargestellt.

Ein Textfile gliedert sich in eine bestimmte Anzahl von sogenannten "Seiten". Jede Seite ist 1024 Bytes groß und nimmt je zwei Blöcke in einem File ein.

Jede Seite besteht aus einer bestimmten Anzahl von Zeilen, die jeweils mit einem "CR"-Zeichen (ASCII-Code 13) abgeschlossen werden. Nun wird hier noch ein Trick angewandt, mit dem Disketteplatz gespart werden soll. In den meisten Texten, vor allem in Pascal-Programmtexten sind die Zeilen eingerückt, d.h. sie beginnen mit einer bestimmten Anzahl von Leerstellen. Würden die Zeilen nun in dieser Form abgespeichert, so bestünde ein Großteil des Files aus Leerstellen. POS kürzt diese Leerstellen sozusagen ab.

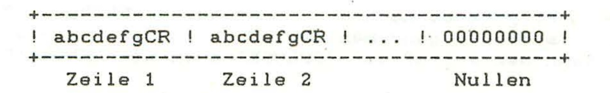
Ist das erste Zeichen in einer Zeile in einem Textfile ein "DLE"-Zeichen (ASCII-Code 16), so bedeutet dies, daß die Zeile eingerückt ist. Der nächste Wert im File gibt nun an, um wieviele Leerstellen. Erst danach beginnt der Inhalt der Zeile. Damit wird relativ viel Platz auf der Diskette eingespart. Eine Zeile, die um zehn Zeichen eingerückt ist, und insgesamt eine Länge von 30 Zeichen hat, nimmt in einem Textfile nur noch 22 Bytes ein.

Wenn eine Seite mit Zeilen gefüllt wird, dann benötigen sie alle zusammen in den wenigsten Fällen genau 1024 Bytes. Der Rest einer Seite, in dem keine Zeile mehr Platz gefunden hat, wird mit Nullen (CHR(0)) aufgefüllt.

Nochmals eine kurze Zusammenfassung zum Aufbau eines Textfiles. Jedes Textfile besteht aus den Editorinformationen und einer Anzahl von Textseiten:

```
+---------------------+------------+-----+-------------+
! Editorinformationen ! Textseite1 ! ... ! Textseite n !
+---------------------+------------+-----+-------------+
      Blöcke 0/1        Blöcke 2/3        Blöcke 2n/2n+1
```

Jede Textseite ist 1024 Bytes groß und gliedert sich in Zeilen. Die restlichen unbenutzen Bytes werden mit Nullen aufgefüllt.



```
+---------------------------------------+
! abcdefgCR ! abcdefgCR ! ... ! 00000000 !
+---------------------------------------+
   Zeile 1     Zeile 2           Nullen
```

Jede Zeile wird mit einem "CR" Zeichen (CHR(13)) abgeschlossen.Handelt es sich um eine Zeile, die eingerückt ist, so ist das erste Zeichen ein "DLE". Danach folgt ein Wert, der angibt, wieviele Leerstellen der Zeile vorangehen. Dieser Wert ist 32 + Anzahl der Leerstellen. Dann folgen die eigentlichen Textzeichen. Es hat sich herausgestellt, daß auch bei den meisten Zeilen, die nicht eingerückt sind, ein "DLE" vorangeht.

```
+-----+------+ +-------------------------+-----+
! DLE ! Wert ! ! Textzeichen abcdefg... ! CR !
+-----+------+ +-------------------------+-----+
 Nur bei ein-
  gerückten
   Zeilen
```

Nun zurück zu unserem Problem. Für unser "READLN", wir nennen es "FREADLN", benutzen wir die "BLOCKREAD"-Funktion. Wir lesen dabei jeweils eine Seite in eine Variable ein. Aus welchem Block in dem File die nächste Seite eingelesen werden muß halten wir in "BLOCK-POINTER" fest. Das Feld "SEITE" enthält nun eine Textseite.

Damit wir wissen, wo die nächste Zeile innerhalb der Seite beginnt, brauchen wir "BUFFERPOINTER". Nach dem Einlesen einer Seite wir "BUFFERPOINTER" auf Null gesetzt.

An "FREADLN" wird ein String ("S") als variables Parameter übergeben. Wir schauen nun, ob das Zeichen, auf das "BUFFERPOINTER" gerade zeigt ein DLE ist. Ist dies der Fall, so schauen wir noch den nächsten Wert an, und spannen vor "S" die entsprechende Anzahl von Leerzeichen.

An der Stelle, auf die "BUFFERPOINTER" nun zeigt, beginnen die Textzeichen der Zeile. Mit der "SCAN"-Funktion stellen wir fest, wo sich das abschließende "CR"-Zeichen der Zeile befindet. Wir wissen nun, wo die Textzeichen der Zeile beginnen, und wo sie enden. Wir müssen noch die einzelnen Zeichen in den String "S" übertragen.

Da es sich bei "SEITE" um ein Feld und bei "S" um einen String handelt, ist eine Übertragung per "COPY" nicht möglich. Eine Schleife, in der die Zeichen einzeln übertragen werden würden, wäre zu langsam. Wir bedienen uns hier der "MOVELEFT"-Prozedur, wobei wir die entsprechende Anzahl von Bytes aus "SEITE" nach "S" übertragen. Damit bei "S" die Längenangabe korrekt wird, füllen wor "S" vorher mit Leerzeichen.

Wir korrigieren nun noch den "BUFFERPOINTER", und haben eine Zeile eingelesen. Da "BLOCKREAD" nur mit unbestimmten Files arbeitet, muß das File, aus dem gelesen werden soll als "FILE" und nicht als "TEXT" deklariert werden.

Zum Öffnen des Files muß die Prozedure "FRESET" benutzt werden. Sie liest die erste Textseite in "SEITE" ein, und setzt "BLOCKPOINTER" und "BUFFERPOINTER" auf Ausgangswerte.

In dem abgedruckten Text gibt es noch die Prozedur "FCLOSE". Sie schließt das File. Im Grunde genommen ist sie nicht als einzelne Prozedur nötig. Man kann den Aufruf von "FCLOSE" auch mit "CLOSE(F)" ersetzen.

Die mit dem Beispielprogramm erreichten Geschwindigkeitssteigerungen sind enorm. Mit der normalen "READLN"-Prozedur braucht das Programm 49.2 Sekunden, um 100 Textzeilen einzulesen. Mit dem abgedruckten Programm waren nur 3,8 Sekunden nötig. Dies ist ein Steigerung um 92.2 Prozent !

Diese Steigerung bringt einige Nachteile mit sich. Zunächt muß für jedes File eine eigene Prozedur geschrieben werden. Es ist in Apple-Pascal nicht möglich, ein File als Parameter an eigene Prozeduren zu

übergeben. Beim eingebauten "READLN" ist dies mit "READLN(F,S)" erlaubt.

Der zweite Nachteil ist, daß vorher bekannt sein muß, wieviele Zeilen sich in dem Textfile befinden. "FREADLN" erkennt nicht, wann der Text zuende ist. Man kann natürlich feststellen, ob über das Fileende hinausgelesen werden soll. Ein Fehler tritt allerdings bei der letzten Seite auf. Handelt es sich um einen Text, der in einer früheren Version länger war, so befinden sich in der letzten Seite noch weitere Textzeichen. Diese werden bei "FREADLN" noch miteingelesen. Man kann dies gut mit dem File "SYSTEM.WRK.TEXT" ausprobieren. Die Information, wieviel Bytes in der letzten Seite gültig sind, steht im Directory (siehe Kap. 2.2). Man müßte also zusätzlich das Directory einlesen, um das Ende des Textes erkennen zu können. Im Kapitel 5.2 wird "FREADLN" auf diese Art verbessert.

Wenn jedoch nur bestimmte Texte eingelesen werden sollen, ist "FREADLN" gut geeignet. Ein Anwendung wäre das Einlesen von Hilfstexten für ein Programm. Hier werden die Wartezeiten während des Diskettenzugriffs minimiert.

Wichtige Variablen und Prozeduren

"F" : File, das zum Einlesen eines Textfile benutzt wird

"SEITE" : Bytefeld, das eine Seite des Textfiles aufnimmt

"BUFFERPOINTER" : Zeigt auf das nächste Zeichen innerhalb von "SEITE", das gelesen wird

"BLOCKPOINTER" : Enthält Nummer des Blocks innerhalb des Textfiles, bei dem die nächste Textseite beginnt

"LEER" : String der 120 Leerzeichen enthält. (Könnte auch als Konstante deklariert werden)

"FRESET" : Öffnet ein Textfile und liest die erste Textseite in "SEITE" ein

"FREADLN" : Liest einen String aus dem Textfile

"FCLOSE" : Schließt das Textfile

Programm "FAST.TEXT"

```
program fast;

var
{ Variablen, die von "FRESET","FRAEDLN" und "FCLOSE" benutzt werden }
          f:file;
         seite:packed array[0..1023] of char;
  bufferpointer:integer;
   blockpointer:integer;
          leer:string[120];
{ Variablen, die von Hauptprogramm benutzt werden }
   s:string;
   i:integer;
```

```
procedure freset(n:string); { Oeffnet das File mit dem Namen 'N' }
var dummy:integer;
begin
  reset(f,n);
  dummy:=blockread(f,seite,2,2); { Erste Seite einlesen }
  bufferpointer:=0;              { Zeichenzeiger auf das erste Zeichen setzen }
  blockpointer:=4;               { Blockzeiger auf die naechste Seite im }
end;                             { File setzen }

procedure freadln(var s:string); { Den String 'S' aus dem File lesen }
var  n,no,start2,dummy:integer;

begin
  s:='';
  if (seite[bufferpointer]=chr(0)) or
     (bufferpointer=1024) then    { Eventuell naechste Seite einlesen }
    begin
      dummy:=blockread(f,seite,2,blockpointer);
      blockpointer:=blockpointer+2;
      bufferpointer:=0;
    end;
  if seite[bufferpointer]=chr(16) { DLE !}
    then begin
           bufferpointer:=bufferpointer+1;
           n:=ord(seite[bufferpointer]); { Anzahl der Blanks holen }
           bufferpointer:=bufferpointer+1;
           s:=copy(leer,1,n); { Blank an den String setzen }
         end;
  no:=scan(1024,=chr(13),seite[bufferpointer]); { Nach CR suchen }
  if no>0
    then
      begin
        start2:=length(s);
        s:=concat(s,copy(leer,1,no));   { String zunaechst mit Blanks fuellen  }
        moveleft(seite[bufferpointer],s[start2+1],no); { Tatsaechliche Zeichen }
        bufferpointer:=bufferpointer+no+1;     { in den String bringen, und    }
      end                                      { Zeichenzeiger korrigieren     }
    else bufferpointer:=bufferpointer+1;
end;

procedure fclose; { File schliessen }
begin
  close(f);
end;

begin
  leer:=('                                        ');
  leer:=concat(leer,leer,leer); { "LEER" mit 120 Blanks fuellen }
  write(chr(7),'##');           { Startsignal zum Zeitmessen    }
  freset('test.text');          { Testfile oeffnen              }
  for i:= 1 to 100 do           { 100 Zeilen einlesen           }
    begin
      freadln(s);
    end;
  write(chr(7),'##');           { Stopsignal zum Zeitmessen     }
  fclose;                       { Testfile schliessen           }
end.
```

2.10 Mehr Platz auf den Disketten

Eine normale POS-Diskette hat 280 Blöcke, d.h. sie kann 140 KByte Daten aufnehmen. Sie ist in 35 Tracks eingeteilt. Dies ist der Fall bei Benutzung der normalen Apple- Laufwerke. DOS und POS sind hier zwar auf 35 Tracks eingestellt, dies ist aber noch nicht die maximale Anzahl von Tracks, die ein Laufwerk hardwaremäßig ermöglicht. So ist bei einem Apple-Laufwerk noch ein 36. Track vorhanden. Er wird aber nicht benutzt und nicht formatiert. Wäre er formatiert, so wäre seine Benutzung kein Problem, und man hätte 4Kbyte mehr Platz auf der Diskette.

Was uns fehlt ist ein Programm, das eine Diskette auch auf 36 Tracks formatieren kann. Das bei Pascal mitgelieferte Programm "FORMATTER" benutzt normalerweise auch nur 35 Tracks. Es ist aber möglich, mit diesem Programm auch 36 oder mehr Tracks zu formatieren.

Wir müssen dazu das File "FORMATTER.DATA" verändern. In diesem File steht eine Assemblerroutine zum Formatieren und ein Datenblock, in dem sich ein "leeres" Directory für eine neuformatierte Diskette befindet. Man kann hier an zwei Stellen eingreifen.

In dem Maschinensprachteil ist eine Routine, die die Diskette Track für Track formatiert. Dabei wird bei Track 0 angefangen und bis Track 35 formatiert. Logischerweise ist 35 ein Abbruchwert für die Formatierung, der auch in der Routine vorkommt. Wenn wir an dieser Stelle einen anderen Wert einsetzen, so gilt dieser als die maximale Anzahl der Tracks. Setzen wir eine 36 ein, so wird auch der bis jetzt ungenutzte 36. Track formatiert. Dieser Abbruchwert befindet sich im 156. Byte des Files.

Wollen wir ihn mit einem Programm ändern, so lesen wir einfach den ersten Block des Files mit "BLOCKREAD" ein, verändern das 156. Byte und schreiben den Block mit "BLOCKWRITE" wieder zurück.

Damit könnten wir aber noch nicht die zusätzlichen 8 Blöcke, die durch den neuen Track entstehen nutzen. Dazu muß noch die Directoryangabe, in der steht, wieviele Blöcke die Diskette hat, geändert werden. Man könnte nach der Formatierung in den Filer gehen und das Z)ero-Kommando benutzen. Man wird dann gefragt, ob sich 280 Blöcke auf der Diskette befinden. Antwortet man mit "N", so kann man einen Wert eingeben. Um den 36. Track mitzubenutzen müßte man "288" eingeben. Dies ist jedoch umständlich, da sich in "FORMATTER.DATA", wie beschrieben, ein "leeres" Directory befindet. Wir müssen nur noch wissen, wo die Angabe über die Anzahl der Blöcke steht, und können hier den entsprechenden Wert einsetzen. Die Anzahl der Blöcke wird in einem 16-Bit- Wert festgehalten, der bei Byte $A0E im File befindet. Diese Bytes stehen im 5. Block des Files. Bei der Änderung dieses Wertes gehen wir wie oben vor.

Das hier abgedruckte Programm übernimmt diese Aufgaben automatisch. Es stellt fest, wieviele Tracks in "FORMATTER.DATA" angegeben sind, und fragt den Benutzer dann nach einer neuen Trackanzahl. Daraufhin wird "FORMATTER.DATA" wie oben beschrieben, verändert.

Man kann nun mit den normalen Formatierprogramm auch Fremdlaufwerke benuzten, die meistens 40 bzw. 41 Tracks haben. Und man kann sich auf den normalen Applelaufwerken 8 Blöcke mehr Platz schaffen, indem man den 36. Track ausnutzt.

Programm "FORMPTCH.TEXT"

```
program formatpatch;

var             f:file;
          block:packed array[0..511] of 0..255;
 tracks,neutracks:integer;

procedure werte_lesen; { Den Wert fuer die Trackanzahl aus }
var dummy:integer;     { FORMATTER.DATA holen, mit dem     }
begin                  { FORMATTER gerade arbeitet.        }
  reset(f,'#4:formatter.data');

  dummy:=blockread(f,block,1,0);
  tracks:=block[156];  { Byte $9B (hex) }

  close(f);
end;

procedure werte_schreiben; { Die neuen Werte fuer die Anzahl }
var dummy:integer;         { der Tracks und der Bloecken in  }
begin                      { FORMATTER.DATA schreiben.       }
  reset(f,'#4:formatter.data');

  dummy:=blockread(f,block,1,0);
  block[156]:=neutracks;  { Byte $9B (hex) }
  dummy:=blockwrite(f,block,1,0);

  dummy:=blockread(f,block,1,5);
  block[14]:=(neutracks*8) mod 256; { Byte $A0E (hex) }
  block[15]:=(neutracks*8) div 256; { Byte $A0F (hex) }
  dummy:=blockwrite(f,block,1,5);

  close(f);
end;

begin
  writeln('----------------------------------------');
  writeln(' Patchen von FORMATTER.DATA');
  writeln('----------------------------------------');
  writeln;
  werte_lesen;
  writeln('Momentan formatiert FORMATTER mit');
  writeln(tracks,' Tracks, d.h. mit ',tracks*8,' Bloecken');
  writeln;
  write('Neue Trackanzahl :');
  readln(neutracks);
  werte_schreiben;
  writeln;
  writeln;
  writeln('FORMATTER formatiert nun mit');
  writeln(neutracks,' Tracks, d.h. mit ',neutracks*8,' Bloecken');
end.
```

3.0 Utilities für Apple-Pascal

Nachdem wir einige Utilities für das Pascal Betriebssystem entwikkelt haben, wenden wir uns nun der Sprache Pascal selber zu. Unser Ziel ist es, einige neue Befehle einzuführen. Daß dabei nicht der Compiler selbst verändert werden kann, sondern nur mit Tricks und neuen Prozeduren gearbeitet werden kann, liegt in der Natur eines Compilers.

Weiterhin werden wir den P-Code Interpreter genauer untersuchen. Dabei soll allerdings nicht auf seine Arbeitsweise eingegangen werden, da diese schon in den Pascal-Handbüchern behandelt wird. Hingegen wird eine Liste vorgestellt, die angibt, in welchen Speicherbereichen welche internen Routinen stehen.

Der Leser sollte bedenken, daß die folgenden Tricks nicht der Pascal-Philosophie folgen. Programme, die sie benutzen sind nicht mehr transportabel und können nicht ohne weiteres auf andere Computer und Compiler übertragen werden. Wer jedoch Programme schreibt, die nur unter Apple-Pascal 1.1 laufen sollen, der also nicht auf Transportabilität achten muß, der sollte diese Tricks ruhig anwenden. Sie erweitern die Grenzen von Pascal, an die man leider sehr oft stoßen kann. Sie gleichen die Lücken aus, die in Apple-Pascal bestehen.

3.1 "PEEK" und "POKE" auch in Pascal

Von BASIC sind die zwei Befehle "PEEK" und "POKE" bekannt. "PEEK" ergibt den Inhalt einer angegebenen Speicherstellen und "POKE" verändert ihn. Im BASIC des Apple II, APPLESOFT, werden die Befehle meistens im Zusammenhang mit der Bildschirmausgabe, der Tastatur und den Paddles verwendet.

Im UCSD Pascal sind diese beiden Befehle nicht vorgesehen. Es ist nicht möglich, z.B. festzustellen, ob ein Knopf an den Paddles gedrückt ist, ohne die "APPLESTUFF"-Unit zu benutzen. (In BASIC wäre die einfach über "PEEK(-16287)" möglich.)

Ein Möglichkeit, eine "PEEK" oder "POKE" Routine zu definieren wäre, ein Maschinensprachen-Programm zu schreiben, an das eine Adresse und eventuell einen Wert übergeben wird. Dazu wären ein Lauf des Assemblers und ein Lauf des Linkers für jedes Programm nötig.

Es geht aber auch in Pascal. Daß dies günstiger ist, ist offenbar: Es fallen der Assembler- und der Linkerlauf weg. Die Pascal Lösung beruht auf der Verwendung von Pointers und Recordvarianten.

Bei Recordvarianten werden in der Variablendefinition verschiedene Stukturen angegeben, die der Record in Abhängigkeit eines Recordfelds annehmen kann. Dieses Feld wird in der Fachsprache "TAGFIELD" genannt. Das sieht z.B. so aus:

```
VAR BEISPIEL: RECORD CASE SCHALTEN:INTEGER OF
                0: (CH:CHAR);
                1: (KETTE:STRING);
                2: (WERT:INTEGER)
              END;
```

Das tatsächliche Aussehen des Records hängt nun vom Wert "BEISPIEL.

Das tatsächliche Aussehen des Records hängt nun vom Wert "BEISPIEL. SCHALTER" ab. Wenn er 1 annimmt, dann ist das zweite Feld des Records der String "KETTE". Wenn "BEISPIEL.SCHALTER" den Wert 2 annimmt, dann ist es das Integer "WERT". Der Compiler reserviert für dieses zweite Feld soviele Bytes, wie die größte Variante benötigt, für "BEISPIEL" also 81 Bytes für "KETTE". Nimmt "BEISPIEL.SCHALTER" 2 an, so bleiben 79 Bytes unbenutzt.

Der zweite Trick bei "PEEK" und "POKE" ist die Verwendung von Pointers. Ein Pointer zeigt auf eine Speicheradresse. Es wird angenommen, daß die dem Pointer zugeordnete Variable dort im Speicher steht. Normalerweise werden diese Pointer nur von einer Variablen auf die andere übergeben, d.h. der eigentliche Wert interessiert nicht und hat nur interne Bedeutung. Es ist aber auch möglich, einem Pointer einen Wert zuzuweisen.

Es ist nun nötig, beide Tricks zu kombinieren. Es ergeben sich folgende Typendefinitionen:

Programm "PEEK-D.TEXT"

```
type
  byte = 0..255;
  spchrinhalt = packed array[0..0] of byte;
  spchrstelle = record case boolean of
                  true: (adresse:integer); { Zeiger }
                  false:(inhalt:^spchrinhalt) { Inhalt }
                end;
```

"BYTE" ist der Wertebereich, den ein Speicherinhalt annehmen kann. Würden wir "BYTE" für den Inhalt einer Speicherzelle nehmen, so erhielten wir falsche Ergebnisse. Denn im UCSD-Pascal beginnen ungepackte Variablen immer an Anfang eines 16-Bit Wortes. Es wäre also nicht möglich, z.B. den Speicherinhalt von 1 zu lesen, da das erste 16-Bit Wort bei 0 beginnt. Wir würden den Inhalt von 0 erhalten.

Dieses Problem besteht nicht bei gepackten Feldern. Sie können überall beginnen. Wir definieren einfach ein gepacktes Feld mit nur einem Element. Mit diesem TYPE "SPCHRINHALT" können wir arbeiten.

Programm "PEEK-P.TEXT"

```
procedure poke(adresse:integer; inhalt:byte);
var dummy:spchrstelle;
begin
  dummy.adresse:=adresse;
  dummy.inhalt^[0]:=inhalt;
end;

function peek(adresse:integer): byte;
var dummy:spchrstelle;
begin
  dummy.adresse:=adresse;
  peek:=dummy.inhalt^[0];
end;
```

'Der Speicher sieht nun so aus:

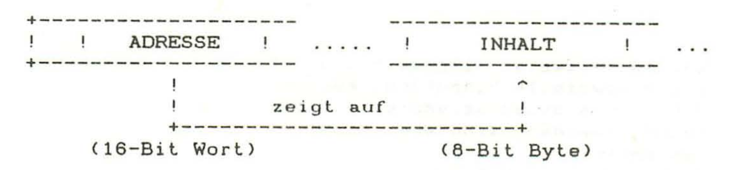


Bild 3.1.1: Wirkung von PEEK/POKE

Anwendungen dieser "PEEK/POKE" Prozeduren finden sich in den folgenden Kapiteln.

3.2 Wo finde ich was ? – eine besondere Liste

Die folgende Liste ist ein Atlas für Apple-Pascal 1.1. Es werden, mit Ausname von Page 0, der Inhalt und die Bedeutung aller Speicherstellen beschrieben. Teilweise sind die hier gegebenen Informationen für den Pascalprogrammierer relativ unwichtig. Wer jedoch mit dem Assembler arbeitet, der findet hier hilfreiche Adressen. Was man mit dieser Liste von Pascal aus anfangen kann, wird in den folgenden Kapiteln exemplarisch dargestellt.

Die Liste wurde aus amerikanischen Publikationen zusammengestellt und durch eigene Untersuchungen vervollständigt. Der Leser sollte sich in diesem Zusammenhang Anhang A und B des POS Handbuchs durchzulesen. Dort ist die Arbeitsweise des P-Code Interpreters beschrieben. Wer weiter interessiert ist, sollte sich das "ATTACH-BIOS document for Apple II Pascal 1.1" von Barry Haynes besorgen. Hier wird das BIOS ausführlich beschrieben.

Die Liste ist nach der Reihenfolge der Speicheradressen geordnet. In der ersten Spalte steht die Anfangsadresse eines Speicherbereiches, in der zweites dessen Ende. Darauf folgt eine Kurzbeschreibung.

Wer sich die einzelnen Routinen genauer anschauen will, der muß sich eines sogenannten Disassemblers bedienen. Er übersetzt die Speicherinhalte in Assemblerbefehle und gibt sie aus. Unter Pascal ist standardmäßig kein Disassembler vorhanden. Wenn der Leser sich keinen eigenen schreiben will, muß er auf den eingebauten Disassembler in den Applesoft-ROMs zurückgreifen. Dazu geht man so vor, daß man zunächst das Pascal-System startet. Apple-Pascal wird nun in die Languagekarte geladen. Man drückt dann auf "RESET". Das System wird versuchen, erneut zu booten. Man kann diesen Vorgang stoppen, indem man wieder "RESET" drückt. Daraufhin befindet man sich im Basic-Interpreter. Mit "CALL-151" wird der eingebaute Monitor aufgerufen.

Nun besteht nur noch das Problem, daß wir uns im ROM-Bereich befinden, und nicht im RAM der Languagekarte. Wir verschieben den gesamten Monitor mit "F800<F800.FFFFM" in die Languagekarte und schalten dann mit "C080 N C080" in den RAM-Bereich um. Nun können wir mit "L" Teile von Apple-Pascal disassemblieren. Um damit etwas anfangen zu können, sind natürlich Maschinensprachkenntnisse notwendig.

Beg.	Ende	Name und Kurzbeschreibung
Zero-Page:		
0	35	Werden zeitweise allgemein benutzt, keine spezielle Bedeutung, können z.B. von eigenen Assemblerroutinen benutzt werden, sind aber nicht beständig.
50	51	BASE: für Base-Prozedur
52	53	MP: Markstack-Pointer
54	55	JTAB: Pointer für Jumptable
56	57	SEG: Segment-Pointer
58	59	IPC: Interpreter Program Counter
5A	5B	NP: New Pointer
5C	5D	KP: Program Stack Pointer
5E	5F	Word-Offeset vom Stack
62	63	Pointer auf Activation Record für Segment #0, Procedure #1 von SYSTEM.PASCAL
64	65	Kopie von BASE
6E	70	Indirekter JMP von Interpreter (JMP (D0xx))
71	73	Indirekter JMP vom CSP-Befehl (JMP (D1xx))
74	75	Temporary
78		Procedure-Nummer, um Activation-Record aufzubauen
7E	7F	MARKSTACK Pointer
80	81	Disk-Puffer
86		Segment-Nummer für Activation Record
8E	8F	Return Adressen
90	91	Pascal-System
96	A2	DLE-Umsetzungstable, von UNITREAD/WRITE ben.
BD	BE	ZEROL/ZEROH: zum Speicherlöschen bei reset
BF	C0	JUMP1/JUMP2: für CASE Anweisung mit chars
C1	C2	BXS1L/BXS1H
C3	C4	BXS2L/BXS2H
C5	C6	CHKPRTL/CHKPRTH
D0	D1	CHECKL/CHECKH
D2	D4	TT1/TT2/TT3
E0		HSMODE: Von Hires-Routinen benutzt
E1		HCMODE: Von Hires-Routinen Benutzt
E2	E3	ACJVAFLD: Pointer auf ATTACH-Kopie des originalen BIOS-Jump-Vektors nach Fold
E4	E5	RTPTR: Pointer auf character device read table
E6	E7	WTPTR: Pointer auf character device write table
E8	E9	UDJVP: Pointer auf user device jump Vektor
EA	EB	DISKNUMP: Pointer auf Disk-Nummer Vektor
EC	ED	JVBFOLD: Pointer auf BIOS-Jump-Vektor vor Fold
EE	EF	JVAFOLD: Pointer auf BIOS-Jump-Vektor nach Fold
F0	F1	BAS1L/BAS1H: Pointer auf 1. Bildschirmseite
F2	F3	BAS2L/BAS2H: Pointer auf 2. Bildschirmseite
F4		CH: horizontale Cursor Position
F5		CV: vertikale Cursor Position
F6	F7	TEMP1/TEMP2
F8	F9	SYSCOM: Pointer auf SYSCOM
100	1FF	6502 Stack
200		BIOS Disk-Puffer
3B1	3FF	Tastatur Puffer
400	7FF	erste Textseite

800	BFF	zweite Textseite
C00		Beginn des Heap (falls nicht verändert)
C40	C7D	INPUT-File Informationen
C7E	CBB	OUTPUT-File Informationen
CBC	CF9	KEYBOARD-File Informationen
CFA	D35	SYSTEM.WRK.TEXT-File Informationen
D36	D71	SYSTEM.WRK.CODE-File Informationen
D72	155E	Zuletzt gelesenes Directory
A988	AC99	Activation-Record für Segment #0, Prozedur #1 von SYSTEM.PASCAL
A994	A995	Pointer auf SYSCOM
A996	A997	Pointer auf INPUT-File Informationen
A998	A999	Pointer auf OUTPUT-File Informationen
A99A	A99B	Pointer auf KEYBOARD-File Informationen
A99C	A9A1	wahrscheinlich unbenutzt
A9A2	A9A3	Pointer auf SYSTEM.WRK.CODE-File Informationen
A9A4	A9A5	Pointer auf SYSTEM.WRK.TEXT-File Informationen
A9A6	A9AB	wahrscheinlich unbenutzt
A9AC	A9AD	Student-Flag vom MISCINFO
A9AE	A9AF	Slow Terminal-Flag vom MISCINFO
A9B0	A9B1	Editor Escape Taste vom MISCINFO
A9B2	A9B3	SYSTEM.WRK.CODE-vorhanden Flag
A9B4	A9B5	SYSTEM.WRK.TEXT-vorhanden Flag
A9B6	A9BD	SYSTEM.WRK.CODE Volume Name (STRING(7))
A9BE	A9C5	SYSTEM.WRK.TEXT Volume Name (STRING(7))
A9C6	A9CD	Volume Name des Workfiles (STRING(7))
A9CE	A9DD	SYSTEM.WRK.CODE Filename (STRING(15))
A9DE	A9ED	SYSTEM.WRK.TEXT Filename (STRING(15))
A9EE	A9FD	Filename des Workfiles (STRING(15))
A9FE	A9FF	Pointer auf freien Heap für Benutzerprogramme
AA02	AA03	Pointer auf KEYBOARD-File Informationen
AA04	AA05	Pointer auf OUTPUR-File Informationen
AA06	AA07	Pointer auf INPUT-File Informationen
AA08	AA0F	Volume Name für Prefix (':') (STRING(7))
AA10	AA17	Volume Name für Boot ('*') (STRING(7))
AA18	AA19	Datum (PACKED RECORD MONAT:0..12; TAG :0..31; JAHR :0..100; END;)
AA1E	AA6F	Kommando Prompt Zeile (STRING(80))
AA70	AA79	Table mit Integer Zehner-Potenzen (ARRAY (0..4) OF INTEGER;)
AA7A	AA85	String mit Nullen für Vertical-Move-Delay vom MISCINFO (STRING(11))
AA86	AA8D	Zahlen-Set (SET of '0'..'9')
AA8E	AB29	Unit Table (ARRAY (0..12) OF RECORD UNITVID: STRING(7);(*Volume*) CASE BLOCKED:BOOLEAN OF TRUE: (LASTBLOCK:INTEGER) END;)
AB2A	AB41	SYSTEM.ASSMBLER Filename (STRING(23))
AB42	AB59	SYSTEM.COMPILER Filename (STRING(23))
AB5A	AB71	SYSTEM.EDITOR Filename (STRING(23))
AB72	AB89	SYSTEM.FILER Filename (STRING(23))
AB8A	ABA1	SYSTEM.LINKER Filename (SRING(23))
ABA2	ABA5	wahrscheinlich unbenutzt
ABA6	ABC5	Configuration Characters vom MISCINFO

ABC6	ABDD	Nächster Filename, der mit SETCHAIN der Unit CHAINSTUFF gesetzt wurde (STRING(23))
ABDE	AC2F	String, der mit SETCVAL der Unit CHAINSTUFF gesetzt wurde (STRING(80))
AC30	AC39	wahrscheinlich unbenutzt
AC3A	AC3B	Lese Exec-File Flag
AC3C	AC3D	Schreibe Exec-File Flag
AC3E	AC3F	Swapping Flag
AC40	AC43	wahrscheinlich unbenutzt
AC44	AC45	'Gerade gebootet' Flag
AC5A	AC61	Volume Name des Exec-Files (STRING(7))
AC62	AC6F	wahrscheinlich unbenutzt
AC70	AC7F	Filename des Exec-Files (STRING(15))
AC80	AC99	wahrscheinlich unbenutzt
AC9A	BD1B	Code von SYSTEM.PASCAL, Segment #0, Prozeduren #29-57
BD1C	BD1D	wahrscheinlich unbenutzt
BD1E	BD5D	Segment Zähler Table des Interpreters
BD5E	BD9E	Segment Location Table des Interpreters
BD9E	BDDD	Segment Prozeduren Verzeichnis des Interpreters
BDDE	BEFF	SYSCOM

BDDE	BDDF	IORESULT: I/O Ergebnis
BDE0	BDE1	XEQERR: Execution Error Code
BDE2	BDE3	SYSUNIT: Unit, von der gebooted wurde
BDE4	BDE5	BUGSTATE: Debugger Status
BDE6	BDE7	GDIRP: Pointer auf zuletzt gelesenes Directory
BDE8	BDE9	Pointer auf EXEC ERROR Activation Record
BDEA	BDEB	STKBASE: Kopie des BASE-Registers des Interpreters
BDEC	BDED	LASTMP: Kopie des MP-Registers des Interpreters
BDEE	BDEF	JTAB: Kopie des JTAB-Registers des interpreters
BDF0	BDF1	SEG: Kopie des SEG-Registers des Interpreters
BDF2	BDF3	BOMBP: Pointer auf das Ende des Programm-Stacks (top of memory)
BDF4	BDF5	BOMIPC: Kopie des IPC, falls ein EXEC ERROR auftritt
BDF6	BDF7	HLTLN: Zeilennummer eines Breakpoints
BDF8	BDFF	BRKPTS: Debugger Breakpoint Table (ARRAY (0..3) OF INTEGER;)
BE00	BE17	wahrscheinlich unbenutzt
BE1C	BE3D	Configuration Informationen vom MISCINFO
		--
BE1C		Lead in from screen
BE1D		Move cursor home
BE1E		Erase to end of screen
BE1F		Erase to end of line
BE20		Move cursor right
BE21		Move cursor up
BE22		Backspace
BE23		Vertical move delay
BE24		Erase line
BE25		Erase screen
BE28	BE29	Screen height
BE2A	BE2B	Screen width
BE2C		Key to move cursor up
BE2D		Key to move cursor down
BE2E		Key to move cursor left
BE2F		Key to move cursor right
BE30		Key to end file

```
BE31          Key for flush
BE32          Key for break
BE33          Key for stop
BE34          Key to delete character
BE35          Non printing character
BE36          Key to delete line
BE37          Editor escape key
BE38          Lead in from keyboard
BE39          Editor accept key
BE3A          Backspace
BE3B  BE3D    wahrscheinlich unbenutzt
BE3E  BEFD    Segment Table (ARRAY (0..31) OF
                                RECORD
                                  UNITNUM :INTEGER;
                                  BLOCKNUM:INTEGER;
                                  CODELENG:INTEGER;
                                END;)
BEFE  BEFF    wahrscheinlich unbenutzt

BF00  BFFF    Interpreter und BIOS-Variablen
              ------------------------------
BF00  BF09    wahrscheinlich unbenutzt
BF0A  BF0D    CONCKVECTOR: Vektor auf die Keyboard-
              Routine
BF0E          SCRMODE: Wenn bit 2 = 1 -> external console
BF0F          LFFLAG: Wenn bit 7 = 0 -> LF'S an Printer
BF10          wahrscheinlich unbenutzt
BF11          NLEFT: Für Kontrolle des horizontalen
              Screen-Scrolls benötigt
BF12          Zähler für Esc-Folgen
BF13  BF14    RANDL/RANDH: Aufgangszahl für Random
BF15          CONFLGS: Flag für Breakbehandlung
              (Bit 7: autofollow, Bit 6: flush, Bit 7: stop)
BF16  BF17    BREAK: Vektor auf Benutzer Break-Routine
BF18          RPTR: Console-Puffer Lesepointer
BF19          WPTR: Console-Puffer Schreibepointer
BF1A  BF1B    RETL/RETH: Return Adresse des Interpreters
              bei BIOS-Aufrufen
BF1C          SPCHAR: Setzt Character Tests
              (Bit 0 = 1 -> nicht auf Ctrl A,Z,K,W,E testen
               Bit 1 = 1 -> nicht auf Ctrl S,F testen)
BF1D  BF1E    IBREAK: Vektor auf Benutzer Break-Routine
BF1F  BF20    ISYSCOM: Pointer auf SYSCOM
BF21          VERSION: 00 = Apple 1.0 / 02 = Apple 1.1
BF22          FLAVOR: 01 = Normales System
                      02 = keine Language-Card (LC)
                      03 = keine LC, keine Sets
                      04 = keine LC, keine Fließkommaroutinen
                      05 = keine LC, keine Sets und keine
                           Fließkommaroutinen
                      06 = LC vorhanden
                      07 = LC, keine Sets
                      08 = LC, keine Fließkommaroutinen
                      09 = LC, keine Sets und keine
                           Fließkommaroutinen
BF23  BF24    Pointer auf BF56
BF25  BF26    wahrscheinlich unbenutzt
BF27  BF2E    SLTTYPES: Table des Slottypen (Art der Karten)
                00 = Karte konnte nicht identifiziert werden
                       (Slot vielleicht leer)
                       01 = Karte konnte identifiziert werden, Code
```

		aber unbekannt
		02 = Disk-Controller
		03 = Kommunikations-Karte
		04 = Serielle Karte
		05 = Printer Karte
		06 = Firmware Karte
BF2F	BF30	XITLOC: Vektor auf XIT Befehl
BF31	BF55	wahrscheinlich unbenutzt
BF56	BF7F	von FORTRAN benutzt
BFC0	BFFF	Boot-Devices, die nicht von Apple sind, können auf diese Speicherstellen zugreifen (z.B. Harddisk)
C000	CFFF	I/O Speicherstellen
D000	DFFF	BIOS-Code (in Bank 2)
D001	D00F	Unterprogramm von DREAD
D010	D027	Unterprogramm von RESINIT
D028		DWRITE: Disk Schreib-Routine
D02C		DREAD: Disk Lese-Routine
D683		DINIT: Disk Init-Routine
D69E		RESINIT: Init-Routine nach einem Hardware-Reset
D772		CONCK: Console Test-Routine
D898		CINIT: Console Init-Routine
D8C6		CREAD: Console Lese-Routine
D8EF		PINIT: Printer Init-Routine
D907		Firmwarekarte Init-Routine
D918		Graphik Init-Routine
D91C		RINIT: Remote Init-Routine
D923		Kommunikationskarte Init-Routine
D930		Serial-Karte Init-Routine
D950		CWRITE: Console Schreib-Routine
D97B		Firmware-Karte Schreib-Routine
D98A		Serial-Karte Schreib-Routine
D99C		RWRITE: Remote Schreib-Routine
D9A4		Printer-Karte Schreib-Routine
D9B2		Kommunikations-Karte Schreib-Routine
D9C3		PWRITE: Printer Schreib-Routine
D9E5		Remote Lese-Routine
D9F8		Kommunikations-Karte Lese-Routine
DA07		Firmware-Karte Lese-Routine
DA15		Serial-Karte Lese-Routine
DCA0		Remote Status und Printer Status Routinen
DCB0		Console Status Routine
DCC5		Disk Status Routine
DCE4		IDSEARCH Routine
DDF5	DE28	Index Table der ersten Buchstaben für IDSEARCH
DE29	DFF9	Identifier Table für IDSEARCH
D000	F22D	Interpreter Code (in Bank 2)
D000	D0FF	Interpreter Jump Table
D100	D151	Jump Table für Standardprozeduren
D152	D154	Jump nach F275
D155	D168	"CHRGET"-Routine für P-Code
D171	D18F	Unterprogramm von LOD und LDA
D190	D1AC	Unterprogramm von MARK und RELEASE
D1EF		EFJ: Equal false jump
D1EF		NFJ: Not equal false jump

D1EF		MATH: CSP 25..31
D1FB	D228	Sichern der Pointer bei EXEC ERROR
D22E	D252	IPC erhöhen
D24D		NOP: No operation
D253		Interpreter Hauptschleife
D25F		FJP: False jump
D267		UJP: Unconditional jump
D296		LDCN: Load constant NIL
D29D		LDCI: Load one word constant
D2A9		SLDL1..SLDL16: Short load local word
D2B6		LDL: Load local word
D2D4		LLA: Load local adress
D2FA		STL: Store local word
D318		SLDO1..SLDO16: Short load global word
D325		LDO: Load global word
D343		LAO: Load global adress
D369		SRO: Store global word
D387		LOD: Load intermediate word
D3AD		LDA: Load intermediate adress
D3DB		STR: Store intermediate word
D401		LDE: Load extended word
D426		STE: store extended word
D44B		LAE: Load extended adress
D467		SIND1..SIND7: Short index and load word
D46A		SINDO: Load indirect word
D47B		STO: Store indirect word
D495		LDC: Load multiply word constant
D4C8		LDM: Load mulitply words
D4F6		STM: Store multiply words
D523		LDB: Load byte
D53D		STB: Store byte
D557		MOV: Move words
D56B		LAND: Logical and
D57E		LOR: Logical or
D591		LNOT: Logical not
D59E		XJP: case jump
D62F		NEW (CSP 1)
D66B		MARK (CSP 32)
D682		RELEASE (CSP 33)
D6A0		XIT: Exit the operation system
D6BB		ABI: Absolute value of integer
D6D9		ADI: Add integers
D6F1		NGI: Negate integers
D703		SBI: Substract integers
D742		MPI: Multiply integers
D789		SQI: Square integers
D839		DVI: Divide integers
D866		MODI: Modulo integers
D87E		CHK: Check against subrange bounds
D8CD		LPA: Load a packed array
D8E5		LSA: Load a constant string adress
D907		SAS: String assign
D948		IXS: Index string array
D96B		IND: Static index and load word
D987		INC: Increment field pointer
D99A		IXA: Index array
D9D9		IXP: Index packed array
DA1C		LDP: Load a packed field
DA72		STP: Store into a packed field
DB20		INT: Set intersection
DB57		DIF: Set difference

DB79		UNI: Set union
DBE5		ADJ: Adjust Set
DC55		INN: Set membership
DCBA		SGS: Built a singleton set
DCCC		SRS: Built a subrange set
DD92	DDD3	Table mit Masken für Manipulation von gepackten Feldern
DDD4		NEQ: Not equal
DDD8		GRT: Greater than
DDDC		LES: Less than
DDE0		GEQ: Greater than or equal
DDE4		LEQ: Less than or equal
DDE8		EQU: Equal
DF2B		LESI: Integer less than
DF2F		GRTI: Integer greater than
DF33		LEQI: Integer less than or equal
DF37		GEQI: Integer greater than or equal
DF3B		NEQI: Integer not equal
DF65		EQUI: Integer equal
E253		CIP: Call intermediate procedure
E2A1		CLP: Call local procedure
E2BD		CGP: Call global procedure
E2D4		CXP: Call external procedure
E2F9		CBP: Call base procedure
E32A		RBP: Return from base procedure
E33F		RNP: Return from non-base procedure
E417		Segement Lese-Routine
E61C		Residentes Segment laden (CSP 21)
E626		Residentes Segment 'ausladen' (CSP 22)
E630		CSP: Call standard procedure
E63A		IDSEARCH (CSP 7)
E640		TREESEARCH (CSP 7)
E6B2		FILLSCHAR (CSP 10)
E6F7		SCAN (CSP 11)
E784		EXIT (CSP 4)
E82B		BPT: breakpoint
E833		HALT (CSP 39)
E841		TIME (CSP 9)
E8A0		MOVELEFT (CSP 2)
E8A0		MOVERIGHT (CSP 3)
E904		MEMAVAIL (CSP 40)
EAC2		ADR: Add reals
EB09		SBR: Substract reals
EB5A		DVR: Divide reals
EC55		MPR: Multiply reals
EC7D		SQR: Square reals
ECB2		ABR: Absolute value of real
ECC0		NGR: negate real
ED3F		FLO: Float next to top-of-stack
ED62		FLT: Floadt top-of stack
EDBB		ROUND (CSP 24)
EDD0		TRUNC (CSP 23)
EDE5		PWROFTEN (CSP 36)
EE0E	EEAA	Table der Zehnerpotenzen
EEAB	EEAC	wahrscheinlich unbenutzt
EEAD	EEBD	Character device write table
EEBD	EECC	Character device read table
EECD		Test auf erlaubte Unit
EEF9		IORESULT (CSP 34)
EF04		IOCHECK (CSP 0)
EF0F		UNITBUSY (CSP 35)

EF1D		UNITWAIT (CSP 37)
EF27		UNITSTATUS (CSP 12)
EFA5		UNITCLEAR (CSP 38)
F069		UNITREAD (CSP 5)
F06E		UNITWRITE (CSP 6)
F22E	FE7B	Code von SYSTEM.PASCAL, Segment #0, Prozeduren #1-28
FE7C	FE7F	wahrscheinlich unbenutzt
FE80	FEAF	Jump Vektoren auf user device
FEC8	FEE8	'Lädt' einen Driver per User-Device-Jump-Vektor
FEE9	FEEE	Start-Routine
FEEF	FEF4	Interrupt, reste und BRK-Routine, springt nach XIT
FEF5	FEFA	Von D756(2) aufgerufen
FEFB	FEFF	Von Hires-Routinen benutzt
FF00	FF41	BIOS Jump Table vor Fold
FF42	FF4E	Schiebt vektor auf Diskettennummer (FEB0) auf Stack
FF4F	FF56	'Lädt' Driver per Disknummer Vektor
FF58		RTS Befehl
FF5C	FF9D	BIOS Jump Table nach Fold
FF5C	FF5E	Vektor auf Console Lese-Routine
FF5F	FF61	Vektor auf Console Schreib-Routine
FF62	FF64	Vektor auf Console Init-Routine
FF65	FF67	Vektor auf Printer Schreib-Routine
FF68	FF6A	Vektor auf Printer Init-Routine
FF6B	FF6D	Vektor auf Disk Schreib-Routine
FF6E	FF70	Vektor auf Disk Lese-Routine
FF71	FF73	Vektor auf Disk Init-Routine
FF74	FF76	Vektor auf Remote Lese-Routine
FF77	FF79	Vektor auf Remote Schreib-Routine
FF7A	FF7C	Vektor auf Remote Init-Routine
FF7D	FF7F	Vektor auf Graphik Schreib-Routine (ruft nur ein RTS Befehl auf)
FF80	FF82	Vektor auf Graphik Init-Routine
FF83	FF85	Vektor auf Printer Lese-Routine
FF86	FF88	Vektor auf Console Status-Routine
FF89	FF8B	Vektor auf Printer Status-Routine
FF8C	FF8E	Vektor auf Disk Status-Routine
FF8F	FF91	Vektor auf Remote Status-Routine
FF92	FF94	Vektor auf Keyboard Test-Routine
FF95	FF97	Vektor auf Routine, die einen Driver per User-Device-Jump Vektor 'lädt'
FF98	FF9A	Vektor auf Routine, die einen driver per Disknummer Vektor 'lädt'
FF9B	FF9D	Vektor auf IDSEARCH
FFEE	FFF5	wahrscheinlich unbenutzt
FFF6	FFF7	Version (0 = Apple 1.1, 1 = Apple 1.0)
FFF8	FFFF	Vektoren
FFF8	FFF9	Start Vektor
FFFA	FFFB	Vektor für Non-Maskable-Interrupt (NMI)

FFFC FFFD Reset Vektor
FFFE FFFF Interrupt request und BRK Vektor

3.3 Was steht wo - eine Anwendung

In diesem Kapitel soll an einem Beispiel erläutert werden, wie man die Informationen, die in der Liste aus Kapitel 3.2 stehen nützen kann. (Wenn im Text auf eine Speicherstelle verwiesen wird, so sollte der Leser diese in der Liste nachschlagen und die Erläuterung lesen.)

Eine Möglichkeit ist, durch gezielte Eingriffe in die systeminternen Routinen das Pascal-System seinen Bedürfnissen anzupassen. So gibt es z.B. die Möglichkeit, die Tastatur des Apple so umzurüsten, daß sie Kleinbuchstaben erzeugen kann. Durch das Einstecken eines neuen Zeichengenerators auf der Hauptplatine können sie dann auch auf dem Bildschirm dargestellt werden. Hierzu ist allerdings ein Eingriff in die Ein- und Ausgaberoutinen des Betriebssytems nötig, da sie auf eine serienmäßige Tastatur eingestellt sind.

Wir wollen in diesem Kapitel jedoch einige systeminterne Variablen einlesen und diese anzeigen. Im Speicherbereich von $A988 bis $BDDD stehen einige solcher Variablen. So z.B. die Filenamen von Assembler, Filers etc., sowie des Workfiles. Auch steht in diesem Bereich, welche Diskette das "Prefix" bedeutet, und von welcher Diskette gebootet wurde. Wir werden den Namen des Workfiles feststellen können, und auch was als "Prefixvolume" (":") und was als "Rootvolume" ("*") gilt.

Weitere interessante Variablen stehen im Bereich von $BF00 bis $BFFF. So z.B., welche Karten in welchen Slots stecken. Wir werden diese Liste ansprechen und diese Informationen anzeigen.

Man kann diese Variablen natürlich nicht nur anzeigen, sondern auch verändern. So kann man von einem Programm aus z.B. die Filer-Funktion "P", die das "Prefixvolume" ändert, ersetzen. So könnte man auch das Programm "PASH" aus Kapitel 2.3 erweitern. Das Setzen des Datums wird im nächsten Kapitel behandelt.

Nun handelt es sich bei diesen Variablen aber nicht nur um einzelne Bytes, sondern z.B. beim Filenamen des Workfiles um einen String mit der Länge 15. Um die Variablen direkt ansprechen zu können müssen wir also den "PEEK/POKE" Mechanismus aus Kapitel 3.1 erweitern. Wenn wir zurückblättern stellen wir fest, daß wir dort einen Record benutzt haben, der es ermöglichte, eine Variable mittels eines Zeigers über eine bestimmte Speicherstelle zu legen. Bei "PEEK/POKE" war diese Variable genau ein Byte groß. Es ist aber genauso gut möglich, das das Record-Feld "SPCHRINHALT" 15 Bytes groß ist. Man kann an dieser Stelle der Record-Definition jeden gültigen Typ angeben. Um den Filename des Workfiles anzusprechen müßte es also lauten "SPRCHINHALT: STRING(15)".

Bei den internen Variablen handelt es sich oft um Strings. Zeiger sind Integer-Werte. Flags bestehen aus einem Boolean-Wert. Es kommen aber auch Felder und Records vor. Das Datum z.B. ist ein gepackter Record (Siehe nächstes Kapitel).

Wir können also durch geeignete Variablendefinitionen jede interne Variable von einem Programm aus ansprechen. Im folgenden Programm werden wir das tun.

Zunächst wollen wir die Volume- und Filenamen des Workfiles anzeigen. In den Speicherstellen $A9B2 und $A9B4 befinden sich Flags, ob überhaupt ein Arbeitstext und -code vorhanden sind. Falls diese Flags den Wert "TRUE" haben stehen die Volumenamen ab $A9CE und die Filenamen ab $A9B6. Zur Erinnerung: Das Workfile muß nicht unbedingt den Namen "SYSTEM.WRK.TEXT", bzw. -".CODE" haben.

Die Volumenamen, für die das ":"- und das "*"-Zeichen stehen, finden sich ab $AA08 als Strings mit der Länge 7. Wir brauchen sie nur anzuzeigen.

Schließlich steht ab $BF27 ein Feld, daß angibt, welche Karte in welchem Slot steckt. Für jeden Slot gibt es eine Byte, das den Kartentyp angibt. Die Bedeutung der Codes ist in der Liste angeben. Wir geben sie mit einer Schleife und einer "CASE"-Anweisung aus.

Man sieht, daß es sehr einfach ist, diese Variablen anzusprechen. Dadurch ergeben sich vielfältige Möglichkeiten. Man kann Operationen, die bis jetzt nur mittels Dienstprogrammen wie dem Filer möglich waren, von seinem Programm aus durchführen. Die Programme werden dadurch viel benutzerfreundlicher. Man braucht sich nur noch in einem Programm auskennen, nicht mehr zusätzlich, z.B. im Filer. Es ist natürlich zu beachten, daß die Programme dann nicht mehr auch andere Pascal-Systeme übertragbar sind. Man sollte hier die Strukturierungsmöglichkeiten von Pascal nutzen, und die Operationen, die systemspezifisch sind in einem Programmblock zusammenfassen. Soll das Programm übertragen werden, so muß nur dieser Teil entsprechend geändert oder ersetzt werden. Am leichtesten sollte eine Übertragung auf ein anderes UCSD-Pascal System sein, da auch hier die gleichen Variablen vorhanden sind. Sie werden aber mit Sicherheit an anderen Stellen stehen.

Programm "WSW.TEXT"

```
program wsw;

type
  byte = packed array [0..0] of 0..255;
  volume_id = string[7];
  file_id = string[15];
  table = packed array [0..7] of 0..255;

var
  flag: record case boolean of
          true : (adresse:integer);
          false: (inhalt :^boolean);
        end;
  volume: record case boolean of
            true : (adresse:integer);
            false: (name:^volume_id);
          end;
  filename: record case boolean of
              true : (adresse:integer);
              false: (name:^file_id);
            end;
  slttyps : record case boolean of
              true : (adresse:integer);
              false: (typ :^table);
            end;
  cnt:integer;
```

```
begin
  writeln(chr(12));
  write('Workfile (text) : ');
  flag.adresse:=-22092;        ( $A9B4 )
  if flag.inhalt^ = true
    then begin
           volume.adresse:=-22082;    ( $A9BE )
           write('''',volume.name^);
           write(':');
           filename.adresse:=-22050; ( $A9DE )
           writeln(filename.name^,'''');
         end
    else writeln('none');
  write('Workfile (code) : ');
  flag.adresse:=-22094;        ( $A9B2 )
  if flag.inhalt^ = true
    then begin
           volume.adresse:=-22090;    ( $A9B6 )
           write('''',volume.name^,':');
           filename.adresse:=-22066; ( $A9CE )
           writeln(filename.name^,'''');
         end
    else writeln('none');

  writeln;

  write('Prefix is ''');
  volume.adresse:=-22008;      ( $AA08 )
  write(volume.name^);
  writeln('''');

  write('Root is    ''');
  volume.adresse:=-22000;      ( $AA10 )
  write(volume.name^);
  writeln('''');
  writeln;

  slttyps.adresse:=-16601;  ( $BF27 )
  for cnt := 0 to 7 do
    begin
      write('Slot #',cnt,' contains ');
      case slttyps.typ^[cnt] of
        00 : writeln('probably nothing');
        01 : writeln('an unknown card');
        02 : writeln('a disk controller');
        03 : writeln('a communications card');
        04 : writeln('a serial card');
        05 : writeln('a printer card');
        06 : writeln('a firmware card')
      end;
    end;

end.
```

3.4 Das Datum

Eine sehr sinnvolle Anwendung der Liste aus Kapitel 3.2 ist eine Prozedur, die das Datum ändern kann. Wenn sie in ein Programm eingebaut wird, kann man es sich sparen, vorher im Filer das Datum zu setzen. Oder man schreibt sie in ein Programm, das als "SYSTEM.STARTUP" auf der Diskette steht. Damit wird schon nach dem Booten das Datum aktualisiert. Vielleicht benutzt man ein Programm täglich. Es kann dann selbstständig das Datum setzen.

Wie wir schon in Kapitel 2.2 gesehen haben, wird das Datum als ein gepackter Record aufgefaßt. Das Format ist im Speicher dasselbe wie im Directory. Aus Kapitel 3.2 wissen wir, wo es im Speicher steht. Mittels eines der schon hinlänglich bekannten Zeiger legen wird diesen gepackten Record über diese Speicherstelle und können so das Datum ansprechen.

Wie bekannt, wird das Datum auch auf Diskette geschrieben, um es beim nächsten Einschalten des Computers wieder zur Verfügung zu haben. Wenn wir das Datum setzen wollen, müssen wir es also auch auf Diskette schreiben. Nun wissen wir aber noch nicht, wohin. Wir schauen uns also den Aufbau des Directorys aus Kapitel 2.2 an. Wenn wir berechnen, wieviel Platz die Informationen vor und nach dem Datumseintrag benötigen, stellen wir fest, daß das 11. Wort im Block 2 einer jeden Diskette das Datum darstellt. Wenn wir es schreiben wollen, müssen wir aber zunächst diesen Block einlesen, und dann das Datum. Dies ist nötig, da ja sonst die Informationen vor und nach dem Datum verloren wären.

Nun zu unserem Programm. Es sollte genau die Funktion erfüllen, die das "D"-Kommando im Filer hat. Im Filer kann man den Tag alleine setzen, indem man das Zeichen "D" vor die betreffende Zahl stellt. Was jedoch nicht möglich ist, nämlich das alleinige Setzen von Monat und Jahr, wollen wir in unserem Programm einbauen. Dazu wird dann der jeweiligen Zahl ein "M" oder ein "Y" vorangestellt. Die Eingabe "M3" würde z.B. den Monat auf März setzen.

Den Monat wollen wir nicht nur durch eine Zahl, sondern wie im Filer durch einen Text ersetzen. Also z.B. "Jan" für Januar. Dazu müssen wir natürlich die Abkürzungen den Monatsnamen haben. Wir schreiben sie in der Prozedur "INIT" in das Feld "MONATE".

Die Prozedur "LESEDATUM" übergibt das Datum an eine Variable, wie oben beschrieben. Wir können es nun ausgeben, wobei wir die Meldungen aus dem Filer übernehmen.

Nun lesen wir einen String ein, der das neue Datum enthält. Dieses wird dann in der Prozedur "CONVERT" ermittelt. Es gibt hier vier Möglichkeiten. Ist das erste Zeichen der Eingabe ein "D", ein "M" oder ein "Y", so soll nur ein Teil des Datums neu gesetzt werden. Es muß dann jeweils der neue Wert aus dem String geholt werden und der entsprechende Teil des Datums verändert werden.

Komplexer ist die Entschlüsselung einer kompletten Datumsveränderung. Hierbei wird angenommen, daß das Datum in der Form "TT-MMM-JJ" vorliegt. D.h., ein Wert für den Tag, darauf ein Bindestrich. Dann der abgekürzte Monatsnamen, wieder ein Bindestrich, und schließlich der Wert für das Jahr. Falls dieses Format an einer Stelle nicht korrekt ist, wird das Datum nicht verändert.

Damit der Benutzer sich nicht absolut an dieses Format halten muß,

werden intern eventuell fehlende führende Nullen eingefügt. Weiterhin wird mit der Prozedur "UPSHIFT" beliebige Groß- und Kleinschreibung ermöglicht und eventuelle Leerstellen entfernt. Die Prozedur "CONVERT" erkennt alle Eingaben, die der Filer auch erkennen würde.

Ist das neue Datum gesetzt, wird es mit der Prozedur "SCHREIBEDATUM" in den Speicher und auf die Diskette gebracht. Die Diskette sprechen wir hier mit "BLOCKREAD/WRITE" an. Dabei benuzten wir nur das Laufwerk von Unit #4.

Bei einer leeren Eingabe nehmen wir an, daß das Datum nicht verändert werden soll.

Das Programm "DATESET" macht also die Datumseingabe einfacher und komfortabler. Die Prozeduren "LESEDATUM" und "SCHREIBEDATUM" können in andere Programm übernommen werden. So kann man z.B. das Programm "PASH" aus Kapitel 2.3 um eine Datumsfunktion erweitern.

Wichtige Typen und Variable

"TDATUM" : Typendefinition für ein Datum

"DATUM" : Variable, die im Programm das Datum aufnimmt

"MONATE" : Feld für die Monatsabkürzungen

Wichtige Prozeduren

"LESEDATUM" : liest das Datum aus den Speicher und übergibt es an eine Variable

"SCHREIBEDATUM" : bringt das an diese Prozedur übergebene Datum in den Speicher und auf die Diskette

"UPSHIFT" : entfernt in dem übergebenen String alle Leerzeichen und wandelt Großbuchstaben in Kleinbuchstaben um

"CONVERT" : setzt das Datum aufgrund der Eingabe

Programm "DATESET.TEXT"

```
program datumsetzen;

type tdatum = packed record
                monat: 0..12;
                tag  : 0..31;
                jahr : 0..99;
              end;

var datum : tdatum;
    monate: array [1..12] of string[3];
    ds    : string;

procedure init; { Initialisiert Monatsnamen }
begin
  monate[ 1]:='Jan';
  monate[ 2]:='Feb';
  monate[ 3]:='Mar';
```

```
    monate[ 4]:='Apr';
    monate[ 5]:='May';
    monate[ 6]:='Jun';
    monate[ 7]:='Jul';
    monate[ 8]:='Aug';
    monate[ 9]:='Sep';
    monate[10]:='Okt';
    monate[11]:='Nov';
    monate[12]:='Dec';
end;

procedure lese_datum(var date: tdatum); { Liest Datum aus Speicher }
var sprchdatum : record case boolean of
                   true : (datum:^tdatum);
                   false: (adresse:integer);
                 end;
begin
  sprchdatum.adresse:= -21992; { Siehe Liste in Kap. 3.2 }
  date:=sprchdatum.datum^;
end;

procedure schreibe_datum (neudatum:tdatum); { Schreibt Datum in Speicher und }
var sprchdatum : record case boolean of     { auf Diskette in Unit 4         }
                   true : (datum:^tdatum);
                   false: (adresse:integer);
                 end;
    diskdatum  : record
                   unused1: packed array [0..9] of integer;
                   datum  : tdatum;
                   unused2: packed array [11..255] of integer;
                 end;
begin

  { Datum in Speicher schreiben }
  sprchdatum.adresse:= -21992;
  sprchdatum.datum^:=neudatum;

  { Datum auf Diskette schreiben }
  unitread(4,diskdatum,512,2,0);     { Block einlesen, um die Felder unused1/2 }
  diskdatum.datum:=neudatum;         { nicht zu veraendern }
  unitwrite(4,diskdatum,512,2,0);    { Block zurueckschreiben }
end;

procedure upshift (t:string; var s:string); { wandelt alle Kleinbuchstaben in }
var i:integer;                              { Grossbuchstaben um und entfernt }
begin
  s:='';
  for i:= 1 to length(t) do
    if ord(t[i])>96
      then begin
             t[i]:=chr(ord(t[i])-32);
             s:=concat(s,copy(t,i,1));
           end
      else if t[i]<>' ' then s:=concat(s,copy(t,i,1));
end;

procedure convert(s:string; var datum:tdatum);
var neudatum:tdatum;
          d:integer;

  function int(t:string):integer; { Wandelt eine zweistellige Zahl in }
```

```
  var i,x:integer;                    { String t in einen Integer-Wert um }
  begin
    x:=0;
    for i:= 1 to 2 do
      x:= 10*x + (ord(t[i])-ord('0'));
    int:=x;
  end;

  function finddatum(t:string):integer; { Wandelt Monatsnamen in die }
  var i:integer;                         { entsprechende Zahl um      }
      s:string;
  begin
    i:=1;
    repeat
      upshift(monate[i],s);
      i:=i+1;
    until (t=s) or (i=13);
    if (i=13) and (t<>s) then finddatum:=0
                          else finddatum:=i-1;
  end;

begin
  if s[1] in ['D','M','Y']
    then if length(s)<3 then insert('0',s,2); { Null einfuegen }
  if s[1]='D' then  { nur Tag veraendern }
                begin
                  d:=int(copy(s,2,2));  { umwandeln }
                  if (d>0) and (d<32) then
                    datum.tag:=d;    { bei gueltigem Tag neu setzen }
                end;
  if s[1]='M' then
                begin
                  d:=int(copy(s,2,2));
                  if (d>0) and (d<32) then
                    datum.monat:=d;
                end;
  if s[1]='Y' then
                begin { nur Jahr veraendern }
                  d:=int(copy(s,2,2));
                  if (d>0) and (d<100) then
                    datum.jahr:=d;
                end;
  if (s[1] in ['1'..'9']) and (length(s)=9)
    then begin
           neudatum:=datum;
           if pos('-',s)=2 then insert('0',s,1); { Null einfuegen }
           d:=int(copy(s,1,2)); { Tag }
           if (d>0) and (d<32) then
             begin
               neudatum.tag:=d;
               if s[3]='-' then
                 begin
                   d:=finddatum(copy(s,4,3)); { Monat }
                   if d>0 then
                     begin
                       neudatum.monat:=d;
                       if s[7]='-' then
                         begin
                           d:=int(copy(s,8,2)); { Jahr }
                           if (d>0) and (d<100) then
                             begin
```

```
                      neudatum.jahr:=d;
                      datum:=neudatum;
                    end;
                  end;
                end;
              end;
            end;
          end;
end;

begin
  init;
  lesedatum(datum);
  writeln('Today is ',datum.tag,'-',
          monate[datum.monat],'-',datum.jahr);
  write('New date ? ');
  readln(ds);
  if ds<>'' then  { Nicht bei leerer Eingabe }
    begin
      upshift(ds,ds);
      convert(ds,datum);
      schreibedatum(datum);
    end;
  writeln('The date is ',datum.tag,'-',
          monate[datum.monat],'-',datum.jahr);
end.
```

3.5 GETKEY - Wir umgehen den Tastaturpuffer

In Basic gibt er den Befehl GET. Es wartet auf die Eingabe eines Zeichens von der Tastatur. In Pascal gibt es eine ähnliche Prozedur, nämlich READ(CH). Sie holt ebenfalls eine Eingabe von der Tastatur. Jedoch ist die Eingabe in Pascal anders organisiert. Es gibt nämlich einen Tastaturpuffer. Das bedeutet, daß jeder Tastendruck zunächst in einem Zwischenspeicher abgelegt wird (dem Puffer), aus dem bei der nächsten Eingabe die Zeichen kommen. Sie können dies ausprobieren, indem Sie z.B. in der Kommandozeile die Zeichenfolge "EIpascal" eintippen. POS wird zunächst den Editor aufrufen. Dann wird, lange nachdem Sie "I" gedrückt haben, der Insert-Modus aufgerufen und darauf das Wort "pascal" in den Text eingegeben. Das kommt durch den Tastaturpuffer, in dem sich Ihre Eingaben befinden, und nacheinander an das Programm gegeben werden.

Oft gibt es aber Situationen, in denen die Eingabe den Benutzer weiterreichende Folgen hat. Durch den Tastaturpuffer ist die Gefahr, daß der Benutzer aus Versehen eine Taste doppelt drückt, vergrößert. Es wäre günstiger, wenn wir den Tastaturpuffer umgehen könnten.

Vor dem gleichen Problem steht man oft bei der Programmierung von Spielen, bei denen es auf die Reaktionsschnelligkeit des Spielers ankommt. Der Tastaturpuffer gibt ihm einen zu großen Zeitvorteil.

Wir brauchen also eine Prozedur, die alle vorhergehenden Tastatureingaben ignoriert und auf eine Eingabe wartet. Das erreichen wir mit der Prozedur GETKEY.

Programm "GETKEY.TEXT"

```
procedure getkey(var key:char);
type                         { TYPE ist lokal zu GETKEY ! }
  byte = 0..255;
  spchrinhalt = packed array[0..0] of byte;
  spchrstelle = record case boolean of
                  true: (adresse:integer);
                  false:(inhalt:^spchrinhalt)
                end;
var dummy:spchrstelle;

begin
  unitclear(1);              { Tastaturpuffer loeschen     }
  dummy.adresse:=-16384;     { $C000 vom Keyboard          }
  repeat                     { Auf Taste warten.           }
  until (dummy.inhalt^[0]>127);
  read(keyboard,key);        { Taste einlesen              }
end;                         { Ab hier wird Tastatur-      }
                             { puffer wieder benutzt !     }
```

Mit "UNITCLEAR(1)" löschen wir den gesamten Tastaturpuffer, und warten dann auf ein Zeichen. Die Tastatur des Apple ist so geschaltet, daß immer dann, wenn eine Taste gedrückt wurde, die Speicherstelle $C000 größer 127 wird. Sonst ist sie kleiner 128.

Mit "PEEK" ist es nun kein Problem, abzufragen wie diese Speicherstelle aussieht. In "GETKEY" ist "PEEK" nicht als einzelne Funktion definiert, die Funktionsweise ist aber genau die gleiche. Wir weisen "DUMMY.ADRESSE" den Wert $C000, oder als Integerzahl, - 16384 zu und warten dann mit einer "REPEAT-UNTIL" Schleife darauf, daß "DUMMY.INHALT^(0)" größer als 127 wird. Tritt dies ein, so ist eine Taste gedrückt worden, und wir können ein Zeichen mittels "READ(KEYBOARD,-KEY)" einlesen, das zugleich an eine Variable Parameter übergeben wird.

Der Aufruf erfolgt mit z.B "GETKEY(CH)".

3.6 Manipulationen am Eingabepuffer

Unter POS ist ein sogenannten "Type-ahead-buffer" implementiert. Er bewirkt, daß von der Tastatur auch dann Zeichen eingelesen werden, wenn keine Eingabe vom Programm erwartet wird. Dies wird so bewerkstelligt, daß bei jeder Programmoperation auch die Tastatur abgefragt wird. Ist ein Taste gedrückt, so wird das entsprechende Zeichen eingelesen. Es wird dann in den sogenannten Tastaturpuffer übernommen. Danach wird im Programm fortgefahren. Wird nun vom Programm eine Eingabe erwartet, wird nachgeschaut, ob sich im Tastaturpuffer noch Zeichen befinden. Ist dies der Fall, so werden diese als Eingabe gewertet. Ist der Tastaturpuffer leer, so kommen die nächsten Eingaben von der Tastatur.

Dies kann man leicht ausprobieren. Man ruft von der Hauptkommandozeile den Filer mit "F" auf. Während der Filer geladen wird, gibt man nun langsam die Zeichen "L",":" und <ret> ein. Wenn der Filer geladen ist, wird nun automatisch eine Directoryliste für die Prefix-Diskette ausgegeben. Dieser Tastaturpuffer hat den Vorteil, daß

keine Eingaben verloren gehen können. Weiterhin muß der Benutzer nicht erst darauf warten, bis Programme geladen, oder Kommandos ausgeführt worden sind.

In diesem Kapitel wird eine Routine vorgestellt, die es ermöglicht, von einem Programm aus ein bestimmtes Zeichen in diesen Tastaturpuffer zu setzen. Der Sinn einer solchen Routine ist es, Programm automatisch aufrufen zu können. So könnte man z.B. in einem selbstgeschriebenen Assembler oder Compiler die Möglichkeit einbauen, nach einem Fehler direkt in den Editor zu gehen. Man braucht dazu nur das Zeichen "E" in den Tastaturpuffer zu schreiben, und danach das Programm zu verlassen. Nach Verlassen des Programms kehrt das System in die Hauptkommandozeile zurück und erwartet eine Eingabe. Da sich im Tastaturpuffer ein "E" befindet, wird dieses als Eingabe gewertet und der Editor aufgerufen.

Um diese Routine zu realisieren, müssen wir uns auf die Assemblerebene begeben. An eine Maschinenroutinen wird ein Zeichen übergeben, das dann von dieser in den Tastaturpuffer geschrieben wird.

Damit wir ein Zeichen in den Tastaturpuffer schreiben können brauchen wir zunächst einige Informationen über den Tastaturpuffer selbst.

Es handelt sich bei dem Puffer um einen Speicherbereich, in dem die eingelesenen Zeichen zwischengespeichert werden. Dieser Bereich befindet sich unter Apple-Pascal bei $3B1 (hex) und hat eine Länge von maximal 78 Zeichen ($4E (hex)).

Um zu wissen, wo in diesem Puffer das nächste Zeichen abgelegt werden muß, gibt es einen Zeiger, "WPTR" (Write PoinTeR). Er befindet sich unter Apple-Pascal bei $BF19 (hex) und ist logischerweise eine Zahl zwischen 0 und 78. Soll nun ein Zeichen in den Tastaturpuffer geschrieben werden, so kommt es an die Speicherstelle, die sich aus Pufferanfang + "WPTR" ergibt.

Falls "WPRT" einmal über das Pufferende hinauszeigt, so wird "WPTR" auf 0 gesetzt, und das nächste Zeichen wird an Anfang des Puffer gespeichert. Man spricht hier von einem Ringpuffer.

Damit Zeichen aus dem Puffer gelesen werden können, gibt es einen zweiten Zeiger, "RPTR" (Read PoinTeR). Die Speicherstelle, aus der das nächste Zeichen gelesen werden muß, ergibt sich aus Pufferanfang + "RPTR". Hier gilt auch, daß dieser Zeiger auf 0 zurück gesetzt wird, wenn er über das Pufferende hinauszeigt.

Es kann nun vorkommen, daß der Puffer voll ist, d.h., daß schon 78 Zeichen abgespeichert sind. Dies ist dann der Fall, wenn "WPRT" nach der Inkrementierung gleich "RPTR" ist. Er "überholt" "WPTR" sozusagen von hinten. Dann wird die Annahme eines neuen Zeichens verweigert. Auf dem Apple wird als Meldung ein Piepston erzeugt.

Wenn wir also ein Zeichen in den Tastaturpuffer schreiben wollen, müssen wir wie folgt vorgehen. Zunächst wird "WPTR" um 1 erhöht. Ist er größer als 78, so wird er auf 0 zurückgesetzt. Dann wird nachgeprüft, ob der Puffer schon voll ist. Dies ist dann der Fall, wenn "WPTR" gleich "RPTR" ist. Trifft dies zu, wird ein Piepston ausgegeben, und die Routine verlassen.

Ist der Puffer noch nicht voll, so wird der neue "WPTR" gesetzt und Das Zeichen in den Puffer an der Speicherstelle Pufferbeginn +

"WPRT" abgelegt. Damit befindet sich das neue Zeichen im Tastaturpuffer.

Wir können nun das Assemblerprogramm schreiben. Es besteht allerdings nicht nur aus den oben genannten Schritten.

Zur Sicherheit retten wir vor dieser Routine alle 6502 Register auf dem Stack. Nach der Routine holen wir sie wieder von Stack, und stellen so den Ausgangszustand wieder her. Es könnte sein, daß diese zwei Schritte überflüssig sind. Um das festzustellen müßte man jedoch detailliert nachprüfen, ob vor dem Aufruf der Routine in den Registern Werte stehen, die hinterher weiterverwendet werden sollen. Man kann eigentlich davon ausgehen, daß man bei Assemblerroutinen alle Register zur freien Verfügung hat; wir speichern die Register nur sicherheitshalber ab.

Zusätzlich gehören zur Assemblerroutine noch zwei Teile, die die Zusammenarbeit mit dem Pascal-Programm übernehmen. Nach dem Aufruf einer Assemblerroutine befinden sich alle Parameter auf dem Stack. Dabei ist das Parameter, das zuerst übergeben wird am "tiefsten" und das, das zuletzt übergeben wird am "höchsten". Wir haben bei unserer Routine nur ein Parameter, nämlich ein Zeichen. Dieses Zeichen ist von Typ "CHAR". Dieser Typ nimmt normalerweise 2 Bytes ein, von denen das höherwertige zuerst auf den Stack geschoben wird. Da wir aber nur 255 Zeichen kennen, ist dieses unbedeutend. Wir holen zunächst mit "PLA" das niederwertigere Byte vom Stack und speichern es zwischen. Damit der Stack nicht in Unordnung gerät ist noch eine "PLA"-Instruktion nötig, die das bedeutungslose zweite Byte vom Stack holt. Damit ist das Parameter vom Pascal-Programm an die Assemblerroutine übergeben.

Beim Aufruf einer Assemblerroutine befindet sich jedoch noch ein Wert auf dem Stack. Er enhält die Adresse, an die nach der Abarbeitung der Assemblerroutine zurückgesprungen werden soll. Sie befindet sich "über" den Parametern oben auf dem Stack. Sie muß zu Beginn des Assemblerprogramms mit zwei "PLA"-Instruktionen eingelesen und zwischengespeichert werden. Bevor das Assemblerprogramm mit "RTS" verlassen werden kann, muß sie mit zwei "PHA"-Instruktionen wieder auf den Stack gelegt werden. Geschieht dies nicht, kehrt das System nicht an die richtige Adresse zurück und stürzt ab.

Damit ist das Assemblerprogramm komplett und kann eingeben und assembliert werden.

Programm "PUSH.TEXT"

```
        .proc pushchar,1   ; 1 Parameter

retl    .equ 0             ; Zwischenspeicher fuer die Ruecksprung-
reth    .equ 1             ; adresse und das Zeichen. ( Speicher-
char    .equ 2             ; plaetze 0 - 35 koennen ja benutzt werden.

cbuflen .equ 04E           ; Laenge des Zeichenpuffers

conbuf  .equ 003B1         ; Adresse des Zeichenpuffers
rptr    .equ 0BF18         ; Lesezeiger des Puffers
wptr    .equ 0BF19         ; Schreibezeiger des Puffers
bell    .equ 0DB1A         ; Gibt ein CHR(7) aus (Klingel)

        pla                ; Ruecksprungadresse von Stack holen und
```

```
         sta retl             ; zwischenspeichern
         pla
         sta reth

         pla                  ; Zeichen vom Stack holen und zwischen-
         sta char             ; speichern.
         pla                  ; (Das Zeichen wird als 16-Bit-Wort ueber-
                              ; geben. Damit ist das zweite Byte
                              ; bedeutungslos.

save     php                  ; Alle 6502-Register auf Stack schieben
         pha                  ; und damit retten.
         txa
         pha
         tya
         pha

         lda char             ; uebergebenes Zeichen

         ldx wptr             ; Zeigt der Zeiger ueber den Puffer
         inx                  ; hinaus ?
         cpx #cbuflen
         bne label1
         ldx #0               ; Wenn ja, am Anfang des Puffers das
                              ; Zeichen ablegen
label1   cpx rptr             ; Ist der Puffer voll ?
         bne bufok
         jsr bell             ; Wenn ja, Klingelzeichen ausgeben und
         jmp restore          ; das Zeichen nicht annehmen

bufok    stx wptr             ; Schreibezeiger erhoehen
         sta conbuf,x         ; Zeichen im Puffer ablegen

restore  pla                  ; Alle 6502-Register vom Stack holen
         tay                  ; und somit den Ausgangszustand
         pla                  ; wiederherstellen
         tax
         pla
         plp

         lda reth             ; Ruecksprungadresse aus dem Zwischen-
         pha                  ; speicher holen und auf den Stack
         lda retl             ; schieben.
         pha

         rts                  ; zurueck zum Pascal-Programm ...

         .end
```

Wir schreiben nun ein kleines Demoprogramm in Pascal. Die Assemblerroutine wird als "EXTERNAL" deklariert. Sie heißt "PUSHCHAR (C:CHAR)". "C" ist das Zeichen, das in den Tastaturpuffer geschrieben werden soll.

Das Programm gibt eine kleine Kommandozeile aus. Der Benutzer kann auswählen, ob er den Editor oder den Filer starten will, oder ob er eine Liste des Directorys ausgeben haben will. Je nachdem werden die entsprefer geschrieben werden soll.

Das Programm gibt eine kleine Kommandozeile aus. Der Benutzer kann

auswählen, ob er dene Directoryliste die Befehlsfolge "FL:<ret>". Daraufhin wird das Programm verlassen, und die Eingaben aus dem Tastaturpuffer abgearbeitet.

Programm "PUSHIT.TEXT"

```
program pushdemo;

var c:char;

procedure pushchar(c:char);         ( Assembler Routine        )
external;

begin
  writeln('E)ditor  F)iler  L)ist of Directory');
  repeat                             ( Auswahl per Tastatur     )
    read(c);
  until c in ['E','e','F','f','L','l'];
  case c of
    'E','e': pushchar('E');          ( Editor Kommando          )
    'F','f': pushchar('F');          ( Filer Kommando           )
    'L','l': begin
               pushchar('F');        ( Filer aufrufen und danach )
               pushchar('L');        ( das L)ist Kommando fuer  )
               pushchar(':');        ( die Prefix Diskette      )
               pushchar(chr(13));
             end
  end;
end. ( Zu diesem Zeitpunkt befinden sich im Zeichenpuffer die )
     ( entsprechenden Kommandos. Sie werden nach Verlassen    )
     ( des Programms so ausgefuehrt, als wenn sie von der     )
     ( Tastatur aus eingegeben worden waeren.                  )
```

Damit man ein lauffähiges Programm erhält, ist es noch nötig, mit dem Linker die Assemblerroutine in das Hauptprogramm einzubinden. Nach der Eingabe von "L" ergibt sich folgender Bildschirmdialog.

```
    LINKING...

APPLE PASCAL LINKER [1.1]
HOST FILE? PUSHIT
OPENING PUSHIT.CODE
LIB FILE? PUSH
OPENING PUSH.CODE
LIB FILE? <ret>
MAP FILE? <ret>
READING PUSHDEMO
READING PUSHCHAR
OUTPUT FILE? PUSHDEMO
LINKING PUSHDEMO # 1
COPYING PROC PUSHCHAR
```

Bild 3.6.1: Dialog beim Linkerlauf

3.7 Ein Maschinencode Monitor für Pascal

Kapitel 3.2 stellte dem Leser eine Liste nützlicher Speicherstellen im Apple-Pascal vor. Ihm ist es aber bis jetzt nur möglich gewesen, bestimmte Speicherplätze anzusprechen und zu verändern. Wir brauchen also einen kleinen "Monitor", d.h. ein Programm, mit dem wir beliebig direkt im Speicher arbeiten können. Bei der Entwicklung wollen wir uns am sogenannten "Autostart-Monitor" orientieren, der bei jedem Apple II+ im ROM fest installiert ist.

Nehmen wir also das "APPLE II REFERENCE MANUAL" zur Hand und schlagen wir auf Seite 59 eine Zusammenfassung der Kommandos dieses Monitors nach.

Wir können nicht alle Funktionen übernehmen, da einige unter POS einerseits schlecht realisierbar wären, und zum anderen kaum benötigt werden. Wir werden in unserem Programm folgende Funktionen des Autostart-Monitors nicht berücksichtigen: das Lesen und Schreiben auf Kassette und das Starten und Disassemblieren von Programmen. Von den übrigen übernehmen wir nur die Addition und Subtraktion zweier Werte.

Das Format der Befehle übernehmen wir vom Autostart-Monitor. Ebenso die Ausgabe mit ihrer 40 Spalten Formatierung. Alle Ein- und Ausgaben sollen in hexadezimaler Schreibweise erfolgen, wobei die Werte nicht durch ein bestimmtes Zeichen (z.B. "$") als hexadezimal gekennzeichenet werden sollen.

Wir haben es also mit den folgenden Befehlen zu tun:

```
adr   Zeigt den Inhalt der Speicherstelle adr an
adr.adr          Zeigt bis zu 8 Speicherzellen ab adr an

adr:val val ..   Schreibt die Werte val ab adr in den Speicher

adr1<adr2.adr3M Verschiebt adr2 bis adr3 nach adr1
adr1<adr2.adr3V Vergleicht adr2 bis adr3 mit adr1 folgende

val+val          Zeigt Summe an
val-val          Zeigt Differenz an

adr ist eine 16 Bit Adresse (0000-FFFF)
var ist ein 8 Bit Wert (00-FF)
```

Wie wir sehen, kommt bei jedem Kommando ein bestimmtes Zeichen in der Eingabe vor, z.B. "+" bei der Addition. Also können wir über diese Zeichen (".",":","M","V","+","-") die Befehle unterscheiden. Falls nur eine Adresse angegeben wird, so handelt es sich um den ersten Befehl.

Der Rest ist relativ einfach. Wir benötigen Routinen, um Adressen und Werte von hexadezimaler Schreibweise in dezimale Integerzahlen umzuwandeln, und eine Routine, um das Ergebnis der Addition bzw. Substraktion hexadezimal auszugeben.

Es ergibt sich das folgende Programm:

Programm "MONITOR.TEXT"

```
program monitor;
type
  byte = 0..255;
  spchrinhalt = packed array[0..0] of byte;
  spchrstelle = record case boolean of
                  true: (adresse:integer);
                  false:(inhalt:^spchrinhalt)
                end;

var
  hexstr,cmd:string;
  bye:boolean;
  point:integer;

procedure poke(adresse:integer; inhalt:byte); { aus Kap 3.1 }
var dummy:spchrstelle;
begin
  dummy.adresse:=adresse;
  dummy.inhalt^[0]:=inhalt;
end;

function peek(adresse:integer): byte;          { aus Kap 3.1 }
var dummy:spchrstelle;
begin
  dummy.adresse:=adresse;
  peek:=dummy.inhalt^[0];
end;

procedure skip; { Leerstellen in Eingabe ueberlesen }
begin
  if point>length(cmd) then exit(skip);
  while cmd[point]=' ' do
    begin
      point:=point+1;
      if point>length(cmd) then exit(skip);
    end;
end;

function getvalue(var value:integer):boolean; { holt ein Byte aus dem         }
var ch:string[1];     { Eingabestring. Bei einem Fehler wird er Funktionswert }
                      { true }
  procedure error;
  begin
    getvalue:=true;
    exit(getvalue);
  end;

begin
  skip;
  ch:=' ';
  getvalue:=false;
  if point>length(cmd) then error;
  ch[1]:=cmd[point];
  if pos(ch,hexstr)=0 then error;
  value:=pos(ch,hexstr)-1;
  point:=point+1;
  if point>length(cmd) then exit(getvalue);
  ch[1]:=cmd[point];
  if pos(ch,hexstr)=0 then exit(getvalue);
```

```
   value:=value*16+pos(ch,hexstr)-1;
   point:=point+1;
 end;

function getadress(var value:integer):boolean; { holt eine Adresse aus }
var ch:string[1];                                { dem Eingabestring }

  procedure error;
  begin
    getadress:=true;
    exit(getadress);
  end;

begin
  skip;
  ch:=' ';
  getadress:=false;
  if point>length(cmd) then error;
  ch[1]:=cmd[point];
  if pos(ch,hexstr)=0 then error;
  value:=pos(ch,hexstr)-1;
  point:=point+1;
  for point:=point to point+3 do
    begin
      if point>length(cmd) then exit(getadress);
      ch[1]:=cmd[point];
      if pos(ch,hexstr)=0 then exit(getadress);
      value:=value*16+pos(ch,hexstr)-1;
    end;
end;

procedure wrbyt(value:integer); { gibt ein Byte in hexadezimaler }
begin                                              { Schreibweise aus }
  write(hexstr[(value div 16)+1],
        hexstr[(value mod 16)+1],' ');
end;

procedure wradr(value:integer); { gibt eine Adresse aus }
begin
  if value>-1 then write(hexstr[((value div 256) div 16)+1],
                         hexstr[((value div 256) mod 16)+1],
                         hexstr[((value mod 256) div 16)+1],
                         hexstr[((value mod 256) mod 16)+1])
              else begin
                     value:=-value-1;
                     write(hexstr[16-((value div 256) div 16)],
                           hexstr[16-((value div 256) mod 16)],
                           hexstr[16-((value mod 256) div 16)],
                           hexstr[16-((value mod 256) mod 16)]);
                   end;
  write('- ');
end;

procedure disrange;      { gibt einen Speicherauszug in hexadezimaler }
var start,ende:integer; { Schreibweise aus }
         fin:boolean;

  procedure error;
  begin
    write(chr(7));
    exit(disrange);
```

```
  end;

begin
  if getadress(start) then error;
  skip;
  if cmd[point]<>'.' then error;
  point:=point+1;
  if getadress(ende) then error;
  if start mod 8 > 0 then wradr(start);
  repeat
    if start mod 8 = 0 then
      begin
        writeln;
        wradr(start);
      end;
    wrbyt(peek(start));
    fin:=(start=ende);
    start:=start+1;
  until fin;
  writeln;
end;

procedure dis; { gibt naechstes Byte aus }
var val:integer;
begin
  if getadress(val) then write(chr(7))
    else begin
           wradr(val);
           wrbyt(peek(val));
           writeln;
         end;
end;

procedure store; { veraendert eine Speicherstelle }
var val,start:integer;

  procedure error;
  begin
    write(chr(7));
    exit(store);
  end;

begin
  if getadress(start) then error;
  skip;
  if cmd[point]<>':' then error;
  point:=point+1;
  while not getvalue(val) do
    begin
      poke(start,val);
      start:=start+1;
    end;
end;

procedure move; { bewegt einen Speicherblock }
var start,ende,dest:integer;
            fin:boolean;

  procedure error;
  begin
    write(chr(7));
```

```
      exit(move);
    end;

  begin
    if getadress(dest) then error;
    skip;
    if cmd[point]<>'<' then error;
    point:=point+1;
    if getadress(start) then error;
    skip;
    if cmd[point]<>'.' then error;
    point:=point+1;
    if getadress(ende) then error;
    repeat
      poke(dest,peek(start));
      fin:=(start=ende);
      dest:=dest+1;
      start:=start+1;
    until fin;
  end;

procedure verify; { vergleicht zwei Speicherbereiche }
var start,ende,dest:integer;
                fin:boolean;

  procedure error;
  begin
    write(chr(7));
    exit(verify);
  end;

begin
  if getadress(dest) then error;
  skip;
  if cmd[point]<>'<' then error;
  point:=point+1;
  if getadress(start) then error;
  skip;
  if cmd[point]<>'.' then error;
  point:=point+1;
  if getadress(ende) then error;
  repeat
    if peek(start)<>peek(dest) then
      begin
        wradr(start);
        wrbyt(peek(start));
        write('(');
        wrbyt(peek(dest));
        writeln(chr(8),')');
      end;
    fin:=(start=ende);
    dest:=dest+1;
    start:=start+1;
  until fin;
end;

procedure add; { addiert zwei Bytes und gibt das Ergebnis aus }
var a1,a2:integer;

  procedure error;
  begin
```

```
    write(chr(7));
    exit(add);
  end;

begin
  skip;
  if cmd[point]='+' then a1:=0
    else begin
           if getvalue(a1) then error;
           skip;
           if cmd[point]<>'+' then error;
         end;
  point:=point+1;
  if getvalue(a2) then error;
  a1:=a1+a2;
  a1:=a1 mod 256;
  wrbyt(a1);
  writeln;
end;

procedure sub; { substrahiert zwei Bytes und gibt das Ergebnis aus }
var a1,a2:integer;

  procedure error;
  begin
    write(chr(7));
    exit(sub);
  end;

begin
  skip;
  if cmd[point]='-' then a1:=0
    else begin
           if getvalue(a1) then error;
           skip;
           if cmd[point]<>'-' then error;
         end;
  point:=point+1;
  if getvalue(a2) then error;
  a1:=a1-a2;
  if a1<0 then a1:=256+a1;
  wrbyt(a1);
  writeln;
end;

begin
  bye:=false;
  hexstr:='0123456789ABCDEF';
  repeat { Hauptschleife }
    write('*');
    readln(cmd);
    point:=1;
    if cmd<>'' then
      if pos(':',cmd)<>0 then store
         else if pos('M',cmd)<>0 then move
           else if pos('V',cmd)<>0 then verify
             else if pos('.',cmd)<>0 then disrange
               else if pos('+',cmd)<>0 then add
                 else if pos('-',cmd)<>0 then sub
                   else if cmd='Q' then bye:=true
                     else dis;
```

```
  until bye;
end.
```

Wichtige Prozeduren

"PEEK","POKE"	: Wurden aus Kapitel 3.1 übernommen
"WRBYT"	: Ein Byte in hexadezimaler Schreibweise ausgeben
"WRADR"	: Ein Adresse in hexadezimaler Schreibweise ausgeben
"STORE"	: Bytes in Speicherbereich schreiben (":" Kommando)
"MOVE"	: Bytes verschieben ("M" Kommando)
"VERIFY"	: Bytes vergleichen ("V" Kommando)
"DISRANGE"	: Speicherbereich ausgeben ("nnnn.mmmm" Kommando)
"ADD"	: addieren zweier Hexzahlen ("+" Kommando)
"SUB"	: substrahieren zweier Hexzahlen ("-" Kommando)
"DIS"	: nächste Speicherzellen ausgeben (bei leerer Eingabe)

Um das Programm nicht unnütz aufzublähen wurde darauf verzichtet, sämtliche Fehleingaben zu erkennen. So wird z.B. die Eingabe "10<" nicht als Fehler erkannt und mit einem Ton quittiert. Vielmehr wird der Speicherinhalt der Adresse $10 ausgegeben und das "<" überlesen. Da dieses Vorgehen keinerlei Schaden in Form von Fehlern oder Programmabbrüche zur Folge hat, werden diese Eingabefehler toleriert.

Noch ein Hinweis zur Benutzung des Monitors. In der Languagekarte, in der POS steht ist der Speicherbereich $D000 bis $DFFF zweimal vorhanden. Man kann natürlich immer nur eine sogenannte Bank benutzten. Auf die andere schaltet man durch Ansprechen bestimmter Speicherstellen um. Da jedoch der P-Code Interpreter in einer Bank steht, und unser Monitor ein Pascal-Programm ist, darf nicht in die andere Bank geschaltet werden. Das Ergebnis wäre ein Systemabsturz.

Das Programm ist gut geeignet für Erweiterungen durch den Leser. Interessante neue Befehle könnten disassemblieren, assemblieren, das o.g. Problem mit der Bankumschaltung beseitigen, Folgen von Bytes suchen und Speicherauszüge in ASCII darstellen. Auch ein Disassembler für P-Code wäre nützlich.

3.8 Wie man einen Reset abfängt

In APPLESOFT-Basic ist es durch "Verbiegen" eines Zeigers möglich, das Drücken der Reset-Taste so abzufangen, daß ein Programm nicht abgebrochen, sondern weitergeführt wird. Wird unter POS die Reset-Taste gedrückt, so führt das System einen Kaltstart durch, d.h. es wird neu gebootet. Das laufende Programm wird abgebrochen, und alle

Daten sind verloren. Eventuell bleiben auf der Diskette offene Dateien bestehen, mit denen nicht weitergearbeitet werden kann. Die Folgen sind also recht verheerend.

In diesem Kapitel wird ein Trick beschrieben, wie man eben dies vermeiden kann. Es wird kein Kaltstart durchgeführt und man kann anhand einer Variablen erkennen, ob Reset gedrückt wurde.

Dabei gehen wir so vor, daß wir zunächst fordern, daß sich das eigentliche Hauptprogramm in einer einzelnen Prozedur befindet. Wir verbiegen vorher den Reset-Vektor auf eine Assemblerroutine, die diese Prozedur mit "EXIT" verläßt. Nun gibt es noch eine Variable, die zunächst auf "TRUE" gesetzt wird. Sie wird nur dann auf "FALSE" gesetzt, wenn das Hauptprogramm korrekt verlassen wird. Dies bedeutet, das die letzte Anweisung im Hauptprogramm z.B. "RESET:=FALSE" lautet. Da sich das Hauptprogramm in einer einzelnen Prozedur befinden, die bei einem Reset verlassen wird, kann man im eigentlichen Hauptprogramm, d.h. dem Teil, der sich zwischen "BEGIN" und "END." befindet, anhand dieser Variablen feststellen, ob Reset gedrückt wurde. Man kann dann Anweisungen aufrufen, die Daten retten.

Den oben genannte Reset-Vektor gibt es auch unter POS, da er vom 6502-Prozessor gefordert wird. Er befindet sich immer in den Speicherstellen $FFFC und $FFFD (hex). Die dort stehenden Werte werden als Adresse aufgefasst. Diese Adresse wird dann vom Prozessor wie bei einem "JMP" angesprungen. Dort sollte dann eine Routine stehen, die einen Reset behandelt. Normalerweise zeigt der Reset-Vektor unter POS auf $FEEF (hex). Die dort stehende Routine veranlasst den beschriebenen Kaltstart. Da POS in der Languagekarte befindet, können wir diesen Zeiger mit "POKE" verändern.

Das nächste Problem ist, wo wir unsere Assemblerroutine unterbringen. Da es kaum ausreichenden freien Speicherplatz gibt, können wir sie nicht an eine bestimmte Adresse schreiben. Wir sich noch zeigen wird, ist dies auch nicht nötig, da die Routine "relokatibel" ist, d.h. sie läuft an jeder beliebigen Speicheradresse. Dies hängt damit zusammen, daß sie keine bestimmten Sprünge innerhalb der Routine benötigt. Es ist also egal, wo sich die Routine befindet; solange der Reset-Vektor auf sie zeigt funktioniert sie auch.

Wir bringen unsere Routine einfach in einer Variablen unter. Damit wir wissen, wo die Variable, und damit auch unsere Assemblerroutine sich im Speicher befindet, deklarieren wir sie als Zeiger und legen sie mit "NEW" auf den sogenannten "Heap". Auf dem Heap befinden sich alle Variablen unter Pascal. Mit "NEW" kommt eine neue Variable hinzu. Man nennt dies dynamische Variablenverwaltung. Die Speicheradresse der Variablen erhalten wir, indem wir einfach den Wert des Zeigers auslesen.

Unsere Maschinenroutine zur Behandlung eines Reset benötigt 18 Bytes. Wir deklarieren die oben genannte Variable als eine Feld mit 18 Bytes. Wir können nun über den Feldindex unsere Routine in die Variable schreiben.

Diese Routine hat eine einzige Aufgabe. Sie soll das Hauptprogramm, das sich ja in einer Prozedur befindet, mit einem "EXIT" verlassen. "EXIT" ist eine Funktion des P-Code Interpreters, und wir können die Adresse dieser Funktion mittels der Liste aus Kapitel 3.2 feststellen. Im "Apple Pascal Operating System" Handbuch sind auf Seite 242 die Parameter beschrieben, die "EXIT" erwartet. Übergeben werden die Segment- und die Prozedurnummer der Prozedur, die verlassen werden

soll. Beide müssen sich vor Aufruf von "EXIT" auf dem Stack befinden, wobei die ersten zwei Bytes auf dem Stack die Prozedurnummer darstellen, und die zweiten zwei die Segmentnummer.

Da wir annehmen, daß sich das Hauptprogramm in der ersten Prozedur im Programm befindet, ist die Segmentnummer 1 und die Prozedurnummer 2. Wenn sich das Hauptprogramm nicht in der ersten Prozedur befinden soll, muß man zunächst die Segment- und Prozedurnummern feststellen. Man macht dies mit der "$L" Compileroption. Hat man diese zwei Werte, setzt man sie entsprechend in die Assemblerroutine ein.

Diese Routine ist nun einfach aufgebaut. Zunächst wird die zweite Speicherbank der Languagekarte eingeschaltet, da sich dort der P-Code Interpreter befindet. Nun wird zunächst die Segment- und dann die Prozedurnummer auf den Stack geschoben. Danach wird "EXIT" bei $E784 (hex) mit "JMP" angesprungen.

Die Pascalprozedur, die dieses Assemblerprogramm installiert baut sich so auf, daß zunächst die Variable, in der sich die Assemblerroutine befinden soll, mit "NEW" auf den Heap gelegt wird. Dann wird der Reset-Vektor so "verbogen", daß er auf diese zeigt. Nun wird die Assemblerroutine in die Variable geschrieben. Damit ist dieser Vorgang beendet.

Es ist vor Verlassen des Programms nötig, den Reset-Vektor wieder auf den ursprünglichen Zustand zu stellen. Wenn man sonst in der Hauptkommandozeile Reset drücken würde, erhielte man einen "EXIT FROM UNCALLED PROCEDURE" Fehler, der sich immer wiederholen würde. Man könnte nur noch den Apple aus- und wieder einschalten.

Ein Programm, in den Reset abgefangen werden soll baut sich wie folgt auf. Zunächst wird der Reset-Vektor verbogen. Nun wird das eigentliche Hauptprogramm als Prozedur aufgerufen. Danach kann mit einer Variablen festgestellt werden, ob das Hauptprogramm mit Reset verlassen wirde. Ist dies der Fall, so kann man weitere Prozeduren aufrufen. Vor Verlassen des Programms muß der Reset-Vektor wieder zurückgestellt werden.

Das Hauptprogramm, das in einer Prozedur enthalten ist, setzt zu Beginn eine Variable auf "TRUE". Sie wird erst in der letzten Anweisung wieder auf "FALSE" gesetzt. Wird zwischendrin das Hauptprogramm mit einem Reset verlassen, so ist die Variable "TRUE", was anzeigt, daß ein Reset vorkam.

In dem hier angedruckten Programm steht das Hauptprogramm in der ersten Prozedur, und wird mit "FORWARD" erst später angegeben. Dies muß, wie oben beschrieben, nicht immer der Fall sein. Die Assemblerroutine muß aber die richtigen Segment- und Prozedurnummern an "EXIT" übergeben.

Programm "RESET.TEXT"

```
program resetdemo;

type byte = packed array[0..0] of 0..255;
var reset:boolean;

procedure hauptprogramm;
  forward;
```

```
procedure resetaendern;
type routine= packed array[0..17] of 0..255;

var resetroutine:^routine;
    x: record case boolean of
         true : (adresse:integer);
         false: (inhalt :^byte)
       end;
    dummy:integer;

begin
  new(resetroutine);

  x.adresse:=-16245;        { 1. Bank einschalten zum Schreiben          }
  x.inhalt^[0]:=0;
  x.inhalt^[0]:=0;

  x.adresse:=-4;            { Reset-Vektor auf 'Resetroutine' verstellen }
  x.inhalt^[0]:=ord(resetroutine) mod 256;
  x.adresse:=-3;
  x.inhalt^[0]:=ord(resetroutine) div 256;

  x.adresse:=-16248;        { 1.Bank gegen Ueberschreiben schuetzen  }
  x.inhalt^[0]:=0;

  { Resetroutine in den Speicher schreiben }

  resetroutine^[ 0]:=141; { STA $C088 ; 2. Bank einschalten }
  resetroutine^[ 1]:=136;
  resetroutine^[ 2]:=192;
  resetroutine^[ 3]:=169; { LDA #$00  ; segment #1 auf Stack schieben }
  resetroutine^[ 4]:=  0;
  resetroutine^[ 5]:= 72; { PHA }
  resetroutine^[ 6]:=169; { LDA #$01 }
  resetroutine^[ 7]:=  1;
  resetroutine^[ 8]:= 72; { PHA }
  resetroutine^[ 9]:=169; { LDA #$00  ; procedure #2 auf Stack schieben }
  resetroutine^[10]:=  0;
  resetroutine^[11]:= 72; { PHA }
  resetroutine^[12]:=169; { LDA #$02 }
  resetroutine^[13]:=  2;
  resetroutine^[14]:= 72; { PHA }
  resetroutine^[15]:= 76; { JMP Exit  ; Exit bei $E784 }
  resetroutine^[16]:=132;
  resetroutine^[17]:=231;
end;

procedure resetnormal;
var x: record case boolean of
         true : (adresse:integer);
         false: (inhalt :^byte)
       end;
    dummy:integer;
begin
  x.adresse:=-16245;    { 1. Bank zum Schreiben schalten }
  x.inhalt^[0]:=0;
  x.inhalt^[0]:=0;

  x.adresse:=-4;        { Reset-Vektor wieder auf $FEEF  }
  x.inhalt^[0]:=239;    { zuruecksetzen                  }
  x.adresse:=-3;
```

```
    x.inhalt^[0]:=254;
    x.adresse:=-16248;   { 1. Bank gegen Ueberschreiben    }
    x.inhalt^[0]:=0;     { schuetzen                       }
end;

procedure hauptprogramm;
var ch:char;

begin
  if reset then
    begin
      writeln;
      writeln('Reset-Error wurde abgefangen !');
    end;
  reset:=true;
  repeat
    write('Bitte eine Taste oder Reset druecken (ESC beendet)');
    read(ch);
  until ch=chr(27);

  reset:=false;
end;

begin
  resetaendern;
  reset:=false;
  repeat
    hauptprogramm;
  until reset=false;
  resetnormal;
end.
```

Für Notizen

4.0 Text und Graphik

Das Apple-Pascal System bietet mit der "Turtlegraphics" Unit hervorragende Möglichkeiten zur Benutzung der Graphik. In keiner anderen Sprache, die es für den Apple gibt (Basic, Forth etc.) sind so klare, einfach zu benutzende Kommandos vorhanden. Es ist praktisch alles vorhanden, was man braucht und das Zusammenfassen von Befehlsfolgen ist durch die Möglichkeiten der Sprache Pascal sehr einfach und flexibel.

Im zweiten Teil dieses Kapitels beschäftigen wir uns mit Text. In Apple-Pascal ist eine Textbearbeitung schon eingebaut. Der vorhandene Texteditor eignet sich grundsätzlich für alle Sorten vor Schriftstücken. Zumeist wird er nur für die Programmeditierung benutzt. Er ist jedoch auch für die Erstellung von Briefen, Artikeln oder auch Büchern geeignet (Das Manuskript zu diesem Buch wurde unter POS geschrieben). Um mit ihm aus dem Apple ein ideales Textsystem zu machen, fehlen allerdings noch einige Elemente der Textverarbeitung. Die hier vorgestellten kleineren Hilfen sind jedoch nur ein kleiner Anriss der Möglichkeiten. Da es unter Pascal ein festes Datenformat für Texte gibt, ist es gut möglich, z.B. in ein Datenbankprogramm eine Schnittstelle zum Texteditor einzubauen. Hier wäre z.B. an eine sogenannte "Mailmerge"-Funktion, d.h. Serienbriefe zu denken. Nun jedoch zu den konkreten Verbesserungen. Wir werden einen kleinen Mangel des Editors beheben und die Größe eines Textes schnell feststellen können.

4.1 Ein Graphikeditor

In Kapitel 2.7 wurde beschrieben, wie wir ein Bild aus hochauflösenden Graphik abspeichern und wieder laden können. Nun ist es aber sehr aufwendig, für jedes Bild ein spezielles Programm zu schreiben, das es zunächst zeichnet und dann auf der Diskette ablegt. Was wir brauchen, ist ein universelles Programm, mit dem wir auf den Bildschirm beliebig zeichnen können. Dieses Kapitel beschreibt einen solchen Graphikeditor.

Wir stellen zunächst wieder unsere Wünsche an das Programm zusammen. Es sollte es ermöglichen, verschiedene geometrische Grundformen in beliebiger Größe darzustellen. Außerdem sollte auch eine Möglichkeit zum "Freihandzeichnen" vorhanden sein. Dann wollen wir natürlich alle Farben benutzen können, die die "Turtlegraphics" zur Verfügung stellen. Schließlich wollen wir auch noch Texte in die Graphik schreiben und Bilder laden und abspeichern können. Die Eingabe sollte möglichst einfach sein.

Fangen wir mit der Eingabe an. Hier bietet sich die Benutzung eines Joysticks an. Wir können so schnell an beliebige Stellen des Bildschirms durch Hebelbewegung "fahren". Außerdem stehen uns damit auch noch die zwei Buttons als praktisches Eingabemittel zur Verfügung. Zum Aufrufen einer Programmfunktion verwenden wir die Tastatur.

An geometrischen Figuren brauchen wir Punkte, Linien, Kreuze, Rechtecke, ausgefüllte Rechtecke (im Programm "Flächen" genannt), Kreise und gefüllte Kreise ("Scheiben"). Mit ihnen lassen sich gute Graphiken erzeugen. Zudem haben sie alle die Eigenschaft, daß sie nur von einem oder zwei Parametern abhängig sind, so z.B. das Rechteck von den Ecken links oben und rechts unten. Im Bild 4.4.1 sind die Formen und ihre Bezugspunkte zu sehen.

```
          X   Punkt

                +
               /
              / Linie
             /
          X


          X-----
          !    !           Rechteck
          !    !
          -----+

          X-----
          !////!           Fläche (gefülltes Rechteck)
          !////!
          -----+

            ---
           :   :
          :     :
          !  X  !          Kreis
          :     +
           :   :
            ---

            ---
           :///:
          ://///:
          !//X//!          Scheibe (gefüllter Kreis)
          ://///+
           :///:
            ---

           TEXTABCDE       Text

     X  = Parameter 1           +  = Parameter 2
```

Bild 4.1.1: Die einzelnen Graphikelemente und ihre Bezugspunkte

Da wir also nur einen oder zwei Punkte als Parameter benötigen, können wir die zwei Buttons am Joystick gut gebrauchen. Mit jedem Knopf wird ein Parameter festgelegt oder gelöscht.

Damit ist auch schon das Gerüst für das Programm festgelegt. In einer Hauptschleife wird zunächst über der Joystick ein Zeichencursor bewegt. Dann werden die Buttons abgefragt und eventuell die Parameter gesetzt oder gelöscht. Schließlich wird bei einem Tastendruck das entsprechende Kommando ausgeführt und die Hauptschleife beginnt von neuem.

Das Abspeichern und Laden, sowie das Festlegen der Zeichenfarbe wird in Menüs erledigt. Dazu schalten wir in den Textmodus. Danach geht es wieder zurück in den Graphikmodus.

Hinzu kommen noch vier weitere Befehle. Je einen zum Einfärben der ganzen Graphikseite in der Zeichenfarbe und zum Löschen. Dann kann dem Benutzer zur Infomation eine Kommandoübersicht ausgegeben werden

und das Programm beendet werden.

Als Zeichencursor wählen wir einen Pfeil, für die Parameter ein geradestehendes und ein diagonales Kreuz. Sie alle werden mit der "DRAWBLOCK"-Prozedur auf den Schirm gebracht. Hierzu brauchen wir die "Turtlegraphics" Unit. Da wir die PADDLE und BUTTON Werte einlesen müssen, kommt dazu noch die "Applestuff" Unit. Und schließlich benötigen wir für Kreise eine Wurzelfunktion, und damit die "Transcend" Unit.

Das Programm gestaltet sich recht einfach. Im Initialisierungsteil werden die Symbole für den Cursor und die Parameter definiert, sowie Variablen auf Anfangswerte gebracht.

In der Prozedur "readin" findet sich die oben genannte Hauptschleife und Prozeduren für die Kommandos. Es ist sehr einfach, neue Befehle hinzuzufügen. Man braucht nur eine entsprechende Prozedur schreiben, und diese über die Tastatureingabe aufzurufen. Falls ein Joystick benutzt wird, der drei Buttons hat, wird es möglich, auch andere geometrische Formen zu verwenden, die drei Parameter benötigen. Hier wären Dreiecke, andere Polygone und Parallelogramme zu nennen. Man müßte hierzu eine dritte Parametervariable einführen und diese bei der Buttoneingabe berücksichtigen.

Auch ein Befehl, der die Graphikseite auf einem Drucker ausgibt wäre sehr nützlich. Da diese bei fast jedem graphikfähigen Drucker anders aussehen wird, wurde bei dem hier abgedruckten Programm darauf verzichtet.

Wichtige Variablen

"CURSOR","PAR1SHAPE","PAR2SHAPE" : Bitfelder, die das Aussehen der Symbole für den Cursor und die Parameter enthalten

"PAR1","PAR2" : Records für die Parameter. "ON" gibt an, ob das Parameter gesetzt ist und "X" und "Y" dessen Position

"VIEWX1","VIEWX2",
"VIEWY1","VIEWY2" : Enthalten die Werte des Bildschirmrandes. Der Cursor kann nicht über sie hinaus und damit aus dem Bild bewegt werden.

"X","Y" : Enthalten die Position des Cursors

"FREIHAND" : Gibt an, ob gerade freihand gezeichnet wird

Kommandoliste

"P" : Punkt
"L" : Linie
"R" : Rechteck
"F" : Fläche
"K" : Kreis
"S" : Scheibe
"T" : Text in die Graphik schreiben

"E" : Schirm löschen
"H" : Schirm mit Zeichenfarbe füllen
"C" : Zeichenfarbe setzen

"D" : Dateiverwaltung (Lesen und Schreiben eines Bildes)

"I" : Kommandoliste ausgeben
"B" : Programm beenden

Programm "GRAFEDIT.TEXT"

```
{$S+}
program grafeditor;

uses turtlegraphics,applestuff,transcend;

type shape1 = packed array [1..7,1..7] of boolean;
  parameter = record
                on :boolean;
                x,y:integer;
              end;
       bild = packed array[0..8191] of 0..255;
    bildvar = record case boolean of
                true: (adresse:integer);
                false:(zeiger:^bild)
              end;


var cursor,par1shape,par2shape:shape1;
    par1,par2:parameter;
    viewx1,viewx2,viewy1,viewy2,
    x,y:integer;
    freihand:boolean;


segment procedure init;

procedure shape(line:integer; var shpe:shape1; inf:string);
var i:integer;
begin
  for i:= 1 to 7 do
    if inf[i]<>' ' then shpe[8-i,line]:=true;
end;


begin
  initturtle;

  { Shapes erstellen }
  fillchar(par1shape,sizeof(par1shape),chr(0));
  fillchar(par2shape,sizeof(par2shape),chr(0));
  fillchar(cursor,sizeof(cursor),chr(0));

  shape(1,par1shape,'X     X');
  shape(2,par1shape,' X   X ');
  shape(3,par1shape,'  X X  ');
  shape(4,par1shape,'       ');
  shape(5,par1shape,'  X X  ');
  shape(6,par1shape,' X   X ');
  shape(7,par1shape,'X     X');

  shape(1,par2shape,'   X   ');
  shape(2,par2shape,'   X   ');
  shape(3,par2shape,'   X   ');
  shape(4,par2shape,'XXX XXX');
```

```
    shape(5,par2shape,'   X   ');
    shape(6,par2shape,'   X   ');
    shape(7,par2shape,'   X   ');

    shape(1,cursor,'XXXXX  ');
    shape(2,cursor,'XX     ');
    shape(3,cursor,'X X    ');
    shape(4,cursor,'X  X   ');
    shape(5,cursor,'X   X  ');
    shape(6,cursor,'     X ');
    shape(7,cursor,'      X');

    { Variablen initialisieren }
    viewx1:=0;
    viewx2:=278;
    viewy1:=6;
    viewy2:=190;

    x:=140;
    y:=96;

    par1.on:=false;
    par1.x :=280;
    par1.y :=0;
    par2.on:=false;
    par2.x :=280;
    par2.y :=0;

    freihand:=false;

end;

procedure home; { loescht Bildschirm }
begin
  write(chr(12));
end;

procedure bell; { gibt akustisches Zeichen }
begin
  write(chr(7));
end;

procedure readin;
var dummy,pad0,pad1:integer;
    finis:boolean;
    ch:char;
    pcol:screencolor;

  procedure drawcursor; { bringt Zeichencursor auf den Bildschirm }
  begin
    drawblock(cursor,2,0,0,7,7,x,y-6,6);
  end;

  procedure xdrawpars;  { loescht Parameter auf dem Bildschirm }
  begin
    drawcursor;
    if par1.on then drawblock(par1shape,2,0,0,7,7,par1.x-3,par1.y-3,6);
    if par2.on then drawblock(par2shape,2,0,0,7,7,par2.x-3,par2.y-3,6);
  end;

  procedure plot(x,y:integer); { setzt einen Punkt }
```

```
begin
  pencolor(none);
  moveto(x,y);
  turnto(0);
  pencolor(pcol);
  move(0);
end;

procedure punkt;  { Kommando 'P' }
var dot:boolean;
begin
  if par1.on and (not par2.on)
    then begin
           xdrawpars;
           plot(par1.x,par1.y);
           par1.on:=false;
           drawcursor;
         end
    else bell;
end;

procedure linie; { zieht eine Linie auf dem Bildschirm }
begin
  if par1.on and par2.on
    then begin
           xdrawpars;
           pencolor(none);
           moveto(par1.x,par1.y);
           pencolor(pcol);
           moveto(par2.x,par2.y);
           par1.on:=false;
           par2.on:=false;
           drawcursor;
         end
    else bell;
end;

procedure kreuz; { zeichnet ein Kreuz auf dem Bildschirm }
var dx,dy:integer;
begin
  if par1.on and par2.on
    then begin
           xdrawpars;
           dx:=par2.x-par1.x;
           dy:=par2.y-par1.y;
           pencolor(none);
           moveto(par1.x+dx,par1.y+dy);
           pencolor(pcol);
           moveto(par1.x-dx,par1.y-dy);
           pencolor(none);
           moveto(par1.x+dy,par1.y+dx);
           pencolor(pcol);
           moveto(par1.x-dy,par1.y-dx);
           par1.on:=false;
           par2.on:=false;
           drawcursor;
         end
    else bell;
end;

procedure rechteck; { zeichnet ein Rechteck auf dem Bildschirm }
```

```
begin
  if par1.on and par2.on
    then begin
           xdrawpars;
           pencolor(none);
           moveto(par1.x,par1.y);
           pencolor(pcol);
           moveto(par2.x,par1.y);
           moveto(par2.x,par2.y);
           moveto(par1.x,par2.y);
           moveto(par1.x,par1.y);
           par1.on:=false;
           par2.on:=false;
           drawcursor;
         end
    else bell;
end;

procedure flaeche; { zeichnet eine Flaeche auf dem Bildschirm }
var ycount:integer;
         p:parameter;
begin
  if par1.on and par2.on
    then begin
           xdrawpars;
           if par1.y > par2.y then begin
                                     p:=par1;
                                     par1:=par2;
                                     par2:=p;
                                   end;
           for ycount:=par1.y to par2.y do
             begin
               pencolor(none);
               moveto(par1.x,ycount);
               pencolor(pcol);
               moveto(par2.x,ycount);
             end;
           par1.on:=false;
           par2.on:=false;
           drawcursor;
         end
    else bell;
end;

procedure kreis; { zeichnet einen Kreis auf dem Bildschirm }
var radius:real;
    i,laenge,winkel,umfang:integer;
begin
  if par1.on and par2.on
    then begin
           xdrawpars;
           radius:=sqrt((par2.x-par1.x)*(par2.x-par1.x) +
                        (par2.y-par1.y)*(par2.y-par1.y));
           umfang:=trunc(2*radius*3.145);
           laenge:=umfang div 20;
           winkel:=-360 div 20;
           pencolor(none);
           moveto(trunc(par1.x+radius),par1.y);
           turnto(180);
           pencolor(pcol);
           turn((180+winkel) div 2);
```

```
            for i:= 1 to 20 do
              begin
                move(laenge);
                turn(winkel);
              end;
            par1.on:=false;
            par2.on:=false;
            drawcursor;
          end
      else bell;
  end;

  procedure scheibe; {.zeichnet eine Scheibe auf dem Bildschirm }
  var p:parameter;
      i,x,y,x2:integer;
  begin
    if par1.on and par2.on
      then begin
            p:=par1;
            kreis;
            xdrawpars;
            y:=0;
            repeat
              y:=y+1;
            until screenbit(p.x,p.y+y) or screenbit(p.x+1,p.y+y);
            for i:= 0 to y do
              begin
                x:=p.x;
                repeat
                  x:=x+1;
                until screenbit(x,p.y+i);
                pencolor(none);
                moveto(x,p.y+i);
                x2:=x;
                x:=p.x;
                repeat
                  x:=x-1;
                until screenbit(x,p.y+i);
                pencolor(pcol);
                moveto(x,p.y+i);
                pencolor(none);
                moveto(x2,p.y-i);
                pencolor(pcol);
                moveto(x,p.y-i);
              end;
            drawcursor;
          end
      else bell;
  end;

  procedure text_eingabe; { bringt einen eingegebenen Text auf den Bildschirm }
  var s:string;
  begin
    if par1.on and (not par2.on)
      then begin
            textmode;
            xdrawpars;
            writeln('Texteingabe');
            writeln('-----------');
            writeln;
            write('Text >');
```

```
            readln(s);
            pencolor(none);
            moveto(par1.x,par1.y);
            wstring(s);
            par1.on:=false;
            grafmode;
            drawcursor;
            home;
          end
    else bell;
end;

procedure schirm_fuellen; { fuellt den Bildschirm in der Zeichenfarbe }
begin
  xdrawpars;
  fillscreen(pcol);
  par1.on:=false;
  par2.on:=false;
  drawcursor;
end;

procedure schirm_loeschen; { loescht den Bildschirm }
var col:screencolor;
begin
  col:=pcol;
  pcol:=black;
  schirm_fuellen;
  pcol:=col;
end;

procedure color_setzen; { setzt Zeichenfarbe }
var ch:char;
begin
  textmode;
  writeln('Farbe setzen');
  writeln('------------');
  writeln;
  writeln('A) white');
  writeln('B) white1');
  writeln('C) white2');
  writeln('D) black');
  writeln('E) black1');
  writeln('F) black2');
  writeln('G) green');
  writeln('H) violet');
  writeln('I) orange');
  writeln('J) blue');
  writeln('K) reverse');
  writeln;
  writeln('U) unveraendert lassen');
  writeln;
  write('Bitte auswaehlen : ');
  repeat
    read(ch);
  until ch in ['A'..'K','U','a'..'k','u'];
  case ch of
    'A','a': pcol:=white;
    'B','b': pcol:=white1;
    'C','c': pcol:=white2;
    'D','d': pcol:=black;
    'E','e': pcol:=black1;
```

```
      'F','f': pcol:=black2;
      'G','g': pcol:=green;
      'H','h': pcol:=violet;
      'I','i': pcol:=orange;
      'J','j': pcol:=blue;
      'K','k': pcol:=reverse
    end;
    grafmode;
    home;
  end;

  procedure datei; { schreibt und liest Bilder auf und von der Diskette }
  var ch:char;
    name:string;
  fehler:boolean;

    procedure pixsave (filename:string; var er:boolean); { aus Kap. 2.7 }
    var pix: file of bild;
          x: bildvar;
    begin
      {$I-}
      filename:=concat(filename,'.FOTO');
      x.adresse:=8192;
      pix^:=x.zeiger^;
      rewrite(pix,filename);
      if ioresult<>0 then er:=true;
      put(pix);
      if ioresult<>0 then er:=true;
      if not er then close(pix,lock);
      if ioresult<>0 then er:=true;
      {$I+}
    end;

    procedure pixload (filename:string; var er:boolean); { aus Kap. 2.7 }
    var pix: file of bild;
          x: bildvar;
    begin
      {$I-}
      filename:=concat(filename,'.FOTO');
      reset(pix,filename);
      if ioresult<>0 then er:=true;
      get(pix);
      if ioresult<>0 then er:=true;
      close(pix,normal);
      if ioresult<>0 then er:=true;
      if not er then begin
                       x.adresse:=8192;
                       x.zeiger^:=pix^;
                     end;
      {$I+}
    end;

  begin
    textmode;
    xdrawpars;
    writeln('Datei-Verwaltung');
    writeln('----------------');
    writeln;
    writeln('L Laden eines Bildes');
    writeln('A Abspeichern eines Bildes');
    writeln;
```

```
    writeln('V Verlassen');
    writeln;
    write('Bitte auswaehlen: ');
    repeat
      read(ch);
    until ch in ['L','l','A','a','V','v'];
    writeln;
    if not (ch in ['V','v']) then
      begin
        writeln;
        write('Filename : ');
        readln(name);
        par1.on:=false;
        par2.on:=false;
      end;
    fehler:=false;
    if ch in ['L','l'] then pixload(name,fehler);
    if ch in ['A','a'] then pixsave(name,fehler);
    if fehler then writeln(chr(13),'Diskettenfehler !');
    write(chr(13),'Bitte eine Taste druecken ');
    read(ch);
    drawcursor;
    grafmode;
    home;
  end;

  procedure info; { gibt Kommanmdoliste aus }
  var ch:char;
  begin
    textmode;
    writeln('Info ueber die Tastaturkommandos');
    writeln('--------------------------------');
    writeln;
    writeln('P Punkt');
    writeln('L Linie');
    writeln('G Kreuz');
    writeln('R Recheck');
    writeln('F Flaeche');
    writeln('K Kreis');
    writeln('S Scheibe');
    writeln('T Text eingeben');
    writeln('M Malen');
    writeln;
    writeln('H ganzen Schirm einfaerben');
    writeln('E ganzen Schirm loeschen');
    writeln('C Zeichenfarbe setzen');
    writeln;
    writeln('D Dateiverwaltung');
    writeln('I Info');
    writeln;
    writeln('B Beenden');
    writeln;
    write('Bitte eine Taste druecken ');
    read(ch);
    grafmode;
    home;
  end;

begin
  pcol:=white1;
  finis:=false;
```

```
  drawblock(cursor,2,0,0,7,7,x,y-6,6);
  repeat
    { Paddle-Eingabe }
    pad0:=paddle(0);
    for dummy:=1 to 3 do;
    pad1:=paddle(1);
    if (pad0>160) or (pad0<94) or (pad1>160) or (pad1<94) then
      begin
        drawcursor;
        x:=x + trunc( (pad0-127) / 33);
        y:=y - trunc( (pad1-127) / 33);
        if x<viewx1 then x:=viewx1;
        if x>viewx2 then x:=viewx2;
        if y<viewy1 then y:=viewy1;
        if y>viewy2 then y:=viewy2;
        if freihand then plot(x,y);
        drawcursor;
      end;
    { Button-Eingabe }
    if button(0) then
      if par1.on
        then begin
               drawblock(par1shape,2,0,0,7,7,par1.x-3,par1.y-3,6);
               par1.on:=false;
             end
        else begin
               drawblock(par1shape,2,0,0,7,7,x-3,y-3,6);
               par1.on:=true;
               par1.x :=x;
               par1.y :=y;
             end;
    if button(1) then
      if par2.on
        then begin
               drawblock(par2shape,2,0,0,7,7,par2.x-3,par2.y-3,6);
               par2.on:=false;
             end
        else begin
               drawblock(par2shape,2,0,0,7,7,x-3,y-3,6);
               par2.on:=true;
               par2.x :=x;
               par2.y :=y;
             end;
    { Tastatur-Eingabe }
    if keypress then
      begin
        read(keyboard,ch);
        case ch of
          'P','p': punkt;
          'L','l': linie;
          'G','g': kreuz;
          'R','r': rechteck;
          'F','f': flaeche;
          'K','k': kreis;
          'S','s': scheibe;
          'T','t': text_eingabe;
          'M','m': freihand:=not freihand;
          'H','h': schirm_fuellen;
          'E','e': schirm_loeschen;
          'C','c': color_setzen;
          'D','d': datei;
```

```
          'I','i': info;
          'B','b': finis:=true
        end;
      end;
  until finis;
end;

begin
  home;
  init;
  readin;
end.
```

Abschießend noch etwas zur Benutzung. Gute Effekte kann man vor allem durch die Farbe "reverse" erzielen. Damit sehen Flächen und Scheiben, die über andere Formen gehen sehr interessant aus. Schaubilder kann man einfach erstellen durch Linien, Rechtecke, Kreise und entsprechenden Beschriftungen.

4.2 SYSTEM.CHARSET wird editiert

Mit der "Turtlegraphics" Unit kann man mit den Prozeduren "WCHAR" und "WSTRING" Textzeichen in eine Graphik schreiben. Die dazu nötigen Zeichendefinitionen stehen in dem File "SYSTEM.CHARSET". Leider wird bei Apple-Pascal kein Dienstprogramm mitgeliefert, mit dem man das Aussehen der Zeichen verändern kann. Dies müßte jedoch relativ einfach sein, da man "SYSTEM.CHARSET" als Datenfile ansprechen kann.

Jedes Zeichen wird aus 56 Punkten zusammengesetzt. Seine Höhe beträgt acht Zeilen und seine Breite sieben Punkte. Demnach besteht jede Zeichendefinition aus acht Bytes, wobei jedes Byte das Aussehen der jeweiligen Zeile innerhalb des Zeichens bestimmt. Innerhalb eines Bytes gibt jedes Bit an, ob ein Punkt gesetzt wird, oder nicht. Da die Zeichen nur sieben Punkte breit sind, wird das achte Bit (b7) nicht benutzt. Wären z.B. alle acht Bytes eines Zeichens mit dem Wert 127 gefüllt, so ergäbe dies ein weißes Rechteck. Dies ist der Fall beim Zeichen 127. Das Zeichen 32 (Leerzeichen) besteht logischerweise aus achtmal dem Wert 0.

Da in "SYSTEM.CHARSET" die Definitionen für 128 Zeichen enthalten sind, kann man sie in ein Feld einlesen, das 128 Elemente hat. Jedes einzelne Element besteht wiederum aus 8 Bytes. Um Platz zu sparen, handelt es sich um ein gepacktes Feld, wie im "Apple Pascal Language" Handbuch auf den Seiten 99 - 100 beschrieben.

Das Dienstprogramm, das hier beschrieben wird, gestaltet sich einfach. Zunächst werden die Zeichendefinitionen aus "SYSTEM.CHARSET" oder einem anderen Zeichensatz-File eingelesen. Dann kann der Benutzer ein Zeichen auswählen, und in diesem jeden Punkt setzen oder löschen. Zum Schluß wird der Zeichensatz wieder unter einem beliebigen Namen auf der Diskette abgespeichert.

Zu Beginn des Programms wird das Aussehen des Bildschirms mit der Prozedur "INIT" gestaltet. Es werden zwei Felder aufgebaut. Eins, um die 128 Zeichen darzustellen und ein Gitter für die vergrößerte Darstellung eines Zeichens.

Dann wird die Prozedur "READINCHARS" aufgerufen. Sie fragt den Benutzer nach dem Namen des zu ladenden Zeichensatzes und liest in

ein. Die Eingabe des Filenamens erfordert eine eigene Prozedur. Da wir uns im Graphikmodus befinden, würde bei einem "READLN"-Kommando keine Rückmeldung auf dem Bildschirm erfolgen. Also brauchen wir die Prozedur "READFILENAME". Sie liest ein Zeichen ein und gibt es in der Graphik aus. Weiterhin wird die "Backspace" Taste (Pfeil links) zum Löschen des letzten Zeichens benutzt. Mit <ret> wird die Eingabe abgeschlossen. Nachdem der Zeichensatz eingelesen wurde, gibt "READINCHARS" ihn auf dem Schirm aus.

Darauf folgt die zentrale Prozedur des Programms, "EDIT". Der Benutzer kann einen blinkenden Cursor in dem Zeichensatz bewegen. Dazu werden die Tasten "I","M","J" und "K" benutzt. Befindet sich der Cursor über dem gewünschten Zeichen, so kann es durch <ret> editiert werden. Wird "F" eingegeben, wird die Prozedur beendet.

Das Editieren eines einzelnen Zeichens geschieht in der Prozedur "EDITONE". Eine Zeichendefinition besteht, wie oben beschrieben, aus acht Bytes. Da wir beim Zeicheneditieren jedoch einzelne Punkte setzen bzw. löschen wollen, rechnen wir die betreffende Zeichendefinition in ein Boolean-Feld um. Damit können wir jeden Punkt einfach ansprechen.

Wir stellen dann das Zeichen vergrößert dar, wobei jeder einzelne Punkt in dem Zeichen ein Rechteck von 15 X 12 Punkten auf dem Bildschirm einnimmt. Nun wird ein zweiter Cursor sichtbar, der wieder mit den Tasten "I","M","J" und "K" bewegt werden kann.
Mit der Leertaste (<spc>) kann der Benutzer den Punkt, über dem der Cursor gerade steht, "umdrehen". D.h., ein weißer Punkt wird schwarz, und ein schwarzer weiß. Mit dem Kommando "R" kann man das gesamte Zeichen invertieren. Falls ein Zeichen völlig neu gestaltet werden soll, kann es mit "L" gelöscht werden. Mit der Eingabe von <ret> wird die Editierung abgeschlossen, und man kann ein anderes Zeichen auswählen.

Wenn "EDIT" mit "F" abgeschlossen wurde, wird der Zeichensatz mit "WRITEOUTCHARS" wieder unter einem beliebigen Namen auf die Diskette geschrieben, und das Programm ist beendet.

Wird der Zeichensatz später in "SYSTEM.CHARSET" umbenannt, so verwenden die "Turtlegraphics" ihn anstandslos. Der Leser sollte dies ausprobieren, indem er einen neuen Zeichensatz gestaltet, und dann das Programm aus Kapitel 4.1 startet. Er kann nun die erzeugten Graphiken mit seinem individuellen Zeichensatz beschriften.

Wichtige Typen und Variablen

"CHARFIELD" : Typ, der der Struktur eines Zeichensatzes entspricht.

"CHARS" : Variable, die den zu editierenden Zeichensatz aufnimmt.

"CHARFILE" : File zum Einlesen und Schreiben des Zeichensatzes.

"ONECHAR" : Booleanfeld aus der Prozedur "EDITONE", das das Aussehen eines Zeichens aufnimmt.

Kommandoliste

```
"I"  : bewegt Cursor nach oben
"M"  : bewegt Cursor nach unten
"K"  : bewegt Cursor nach rechts
"J"  : bewegt Cursor nach links

<ret>: ein Zeichen editieren
dann: "I","M","K","J" : Cursorbewegungen wie oben
                <spc> : invertiert einen Punkt
                 "R"  : invertiert das ganze
                        Zeichen
                 "L"  : löscht das ganze Zeichen
                <ret> : beendet

"F" : Beendet Editierung
```

Programm "CHAREDIT.TEXT"

```
{$S+}
program charedit;

uses turtlegraphics,applestuff;

type
  charfield = packed array[0..127,0..7] of 0..255;
var
  chars: charfield;
  charfile: file of charfield;

procedure init; { legt Bildschirmlayout fest }
var i:integer;
begin
  initturtle;
  pencolor(none);
  for i := 0 to 8 do
    begin
      { horizontal }
      pencolor(none);
      moveto(0,191-i*16);
      pencolor(green);
      moveto(98,191-i*16);
      { vertikal }
      if i<8 then
        begin
          pencolor(none);
          moveto(i*14,191);
          pencolor(green);
          moveto(i*14,63);
        end;

    end;

  pencolor(none);
  moveto(130,191);
  pencolor(white);
  moveto(278,191);
  pencolor(white1);
  moveto(278,115);
  pencolor(white);
```

```
  moveto(130,115);
  pencolor(white1);
  moveto(130,191);

end;

procedure read_filename(var inp:string); { liest einen Filenamen von der }
var ret:boolean; { Tastatur ein, und bestaetigt jeden Buchstaben auf dem }
      c:char;                                               { Bildschirm }
begin
  inp:='';
  pencolor(none);
  moveto(0,0);
  wstring('Zeichensatz?');
  ret:=false;
  repeat
    moveto(90+length(inp)*7,0);
    wchar('_');
    repeat
    until keypress;
    read(c);
    if eoln then ret:=true
      else if c=chr(8) then
                          if length(inp)>0
                            then begin
                                   moveto(90+length(inp)*7,0);
                                   wchar(' ');
                                   inp:=copy(inp,1,length(inp)-1);
                                 end
                            else write(chr(7))
        else begin
               moveto(90+length(inp)*7,0);
               wchar(c);
               inp:=concat(inp,' ');
               inp[length(inp)]:=c;
             end;
    if ret then if length(inp)=0 then
      begin
        write(chr(7));
        ret:=false;
      end;
  until ret;
  moveto(90+length(inp)*7,0);
  wchar(' ');
  readln;
end;

procedure readin_chars; { liest Zeichensatz von der Diskette ein }
var name:string;
    i,j:integer;

begin
  read_filename(name);
  reset(charfile,name);
  get(charfile);
  chars:=charfile^;
  close(charfile);
  moveto(0,0);
  wstring('                                        ');

  for i:=0 to 7 do
```

```
    for j:=0 to 15 do
      drawblock(chars[i*16+j],1,0,0,7,8,134+j*9,181-i*9,10);
end;

procedure edit; { laesst den Benutzer den Zeichensatz editieren }
var setx,sety:integer;
      ed,quit:boolean;
           ch:char;
       cursor:packed array[0..7,0..6] of boolean;
     cursoron:boolean;
        timer:integer;


  procedure editone(charno:integer); { laesst den Benutzer ein Zeichen }
  var shape:packed array[1..15,1..12] of boolean;          { editieren }
      onechar: packed array [0..7,0..6] of boolean;
      i,j:integer;
      x,y:integer;
   byte,p:integer;
     quit:boolean;
       ch:char;

    procedure giveout;  { gibt ein Zeichen vergroessert aus }
    var xcnt,ycnt:integer;
    begin
      for ycnt:= 0 to 7 do
        for xcnt:=0 to 6 do
          if onechar[ycnt,xcnt]
            then drawblock(shape,2,0,0,12,15,2+xcnt*14,64+ycnt*16,15)
            else drawblock(shape,2,0,0,12,15,2+xcnt*14,64+ycnt*16, 0);
    end;

  begin

    for i:= 0 to 7 do
      begin
        byte:=chars[charno,i];
        byte:=byte mod 128;
        p:=64;
        for j:= 0 to 6 do
          begin
            if byte div p = 1
              then onechar[i,6-j]:=true
              else onechar[i,6-j]:=false;
            byte:=byte mod p;
            p:=p div 2;
          end;
      end;

    giveout;

    moveto(0,40);
    wstring('I,M,K,J  bewegen den Cursor      ');
    moveto(0,30);
    wstring(' <spc>   invertiert einen Punkt ');
    moveto(0,20);
    wstring('   R     invertiert das Zeichen ');
    moveto(0,10);
    wstring('   L     loescht das Zeichen     ');
    moveto(0,0);
    wstring(' <ret>   beendet Zeicheneditierung');
```

```
x:=0;
y:=0;
quit:=false;
chartype(6);
moveto(4+x*14,67+y*16);
wchar(chr(127));
repeat
  repeat
  until keypress;
  read(ch);
  moveto(4+x*14,67+y*16);
  wchar(chr(127));
  case ch of
    'i','I': begin
               y:=y+1;
               if y>7 then y:=0;
             end;
    'm','M': begin
               y:=y-1;
               if y<0 then y:=7;
             end;
    'k','K': begin
               x:=x+1;
               if x>6 then x:=0;
             end;
    'j','J': begin
               x:=x-1;
               if x<0 then x:=6;
             end;
        ' ': begin
               if not eoln then
                 begin
                   onechar[y,x]:=not onechar[y,x];
                   if onechar[y,x]
                     then drawblock(shape,2,0,0,12,15,2+x*14,64+y*16,15)
                     else drawblock(shape,2,0,0,12,15,2+x*14,64+y*16, 0);
                 end;
             end;
    'r','R': begin
               for i:=0 to 7 do
                 for j:=0 to 6 do
                   onechar[i,j]:=not onechar[i,j];
               giveout;
             end;
    'l','L': begin
               fillchar(onechar,sizeof(onechar),chr(0));
               giveout;
             end
  end;
  if eoln then begin
                 readln;
                 quit:=true;
                 moveto(4+x*14,67+y*16);
                 wchar(chr(127));
               end;
  moveto(4+x*14,67+y*16);
  wchar(chr(127));
until quit;

for i:= 0 to 7 do
  begin
```

```
          byte:=0;
          p:=64;
          for j:= 0 to 6 do
            begin
              if onechar[i,6-j] then byte:=byte+p;
              p:=p div 2;
            end;
          chars[charno,i]:=byte;
        end;

      fillchar(onechar,sizeof(onechar),chr(0));
      giveout;
    end;


  begin
    setx:=1;
    sety:=1;
    fillchar(cursor,sizeof(cursor),chr(255));
    drawblock(cursor,1,0,0,7,8,125+setx*9,190-sety*9,6);

    repeat
    chartype(10);
      moveto(0,40);
      wstring('                                    ');
      moveto(0,30);
      wstring('                                    ');
      moveto(0,20);
      wstring('I,M,K,J  bewegen den Cursor         ');
      moveto(0,10);
      wstring(' <ret>   waehlt ein Zeichen aus');
      moveto(0,0);
      wstring('   F     beendet die Editierung      ');
      quit:=false;
      repeat

        cursoron:=true;
        timer:=300;

        repeat { wartet auf Tastendruck und gibt blinkenden Cursor aus }
          timer:=timer-1;
          if timer=0 then
            begin
              timer:=300;
              drawblock(cursor,1,0,0,7,8,125+setx*9,190-sety*9,6);
              cursoron:=not cursoron;
            end;
        until keypress;
        if not cursoron then
          drawblock(cursor,1,0,0,7,8,125+setx*9,190-sety*9,6);

        read(ch);
        if ch in ['i','I','m','M','k','K','j','J'] then
          begin
            drawblock(cursor,1,0,0,7,8,125+setx*9,190-sety*9,6);
            case ch of
              'i','I':begin
                        sety:=sety-1;
                        if sety<1 then sety:=8;
                      end;
              'm','M':begin
```

```
                          sety:=sety+1;
                          if sety>8 then sety:=1;
                        end;
              'k','K':begin
                          setx:=setx+1;
                          if setx>16 then setx:=1;
                        end;
              'j','J':begin
                          setx:=setx-1;
                          if setx<1 then setx:=16;
                        end
            end;
            drawblock(cursor,1,0,0,7,8,125+setx*9,190-sety*9,6);
          end;
        if ch in ['f','F'] then begin
                                  quit:=true;
                                  ed:=false;
                                end;
        if eoln then begin
                       readln;
                       quit:=true;
                       ed:=true;
                     end;
      until quit;
      if ed then begin
                   editone((sety-1)*16+(setx-1));
                   drawblock(cursor,1,0,0,7,8,125+setx*9,190-sety*9,6);
                   drawblock(chars[(sety-1)*16+(setx-1)],
                             1,0,0,7,8,125+setx*9,190-sety*9,10);
                   drawblock(cursor,1,0,0,7,8,125+setx*9,190-sety*9,6);
                 end;
  until ed=false;
  drawblock(cursor,1,0,0,7,8,125+setx*9,190-sety*9,6);
end;

procedure writeout_chars; { schreibt Zeichensatz auf Diskette }
var name:string;
begin
  moveto(0,20);
  wstring('                                   ');
  moveto(0,10);
  wstring('                                       ');
  moveto(0,0);
  wstring('                                       ');
  read_filename(name);

  rewrite(charfile,name);
  charfile^:=chars;
  put(charfile);
  close(charfile,lock);

end;

begin
  init;
  readin_chars;
  edit;
  writeout_chars;
  write(chr(12));
end.
```

4.3 Wieso nicht Lores-Graphik ?

Apple-Pascal kennt mit der "Turtlegraphics" Unit nur die hochauflösende Graphik mit 8 Farben, wie unter Basic. Applesoft kennt aber auch die Blockgraphik (Lores-Graphik) mit 16 Farben. Da diese in Basic möglich ist, müsste es sie eigentlich auch in Pascal geben. Wir wollen uns eine Unit schreiben, mit der wir Zugriff auf die Blockgraphik erhalten.

Um die Informationen zu bekommen, wie die Blockgraphik angesteuert wird, brauchen wir das "Apple II Reference Manual", das mit jedem Apple ausgeliefert wird. Angaben über die Loresgraphik finden wir auf den Seiten 17, 18 und 130.

Die Loresgraphik wird eingeschaltet, indem die Speicherstellen -16304 und - 16298 angesprochen werden. Die erste schaltet bei Ansprache in den Graphikmodus und die zweite schaltet auf Blockgraphik. Der Speicherbereich, der angibt, was in der Graphik zu sehen ist, liegt parallel zur Bildschirmseite, die bei Speicherstelle 1024 beginnt. Anstatt eine Zeichens werden jedoch zwei Punkte angezeigt. Ihre Farben hängen von Inhalt der jeweiligen Speicherstelle ab. In jedem Byte des Speicherbereiches sind also die Farben für zwei Punkte in der Loresgraphik enthalten. Da ein Byte 256 Werte annehmen kann, gibt es für jeden Punkt in der Blockgraphik 16 Farben, da ein halbes Byte (mit 4 Bits) 2 hoch 4 = 16 Werte annehmen kann. Ein halbes Byte wird Nibble genannt.

In Textmodus können 24 Zeile zu je 40 Zeichen dargestellt werden. Da in der Blockgraphik jedes Zeichen zwei übereinanderliegende Punkte ergibt, beträgt die Auflösung 40 * 48 Punkte. Es bleibt nur noch offen, welches Nibble in einer Speicherstelle des Bildschirmspeichers welchen Punkt beeinflußt. Das erste Nibble (Bits 0-3) gibt die Farbe für den oberen Punkt und das zweite (Bits 4-7) die des unteren an.

```
         Textmodus         !       Blockgraphik
-------------------------------------------------
                           !
         +-----+           !         +-----+
         !  *  !           !         !/////!
         ! * * !           !         !/////!  A
         !*   *!           !         !/////!
         !*****!           !         !/////!
         !*   *!           !         !=====!
         !*   *!           !         !=====!  B
         !     !           !         !=====!
         +-----+           !         +-----+
                           !
        1 Zeichen          !        2 Punkte
                           !
     +---------------+     !    +---------------+
     !b!b!b!b!b!b!b!b!     !    !b!b!b!b!b!b!b!b!
     +---------------+     !    +---------------+
                           !
     7               0     !    7     4 3      0
                           !
     Ganzes Byte be-       !    Bits 7-4 bestimmen
     stimmt Zeichen        !    Farbe für Punkt A,
                           !    Bits 3-0 für Punkt B
```

Bild 4.3.1: Darstellung eines Punktes in der Lores-Graphik

Mit den PEEK und POKE Prozeduren aus Kapitel 3.1 sollte es uns nun möglich sein, Punkte in der Blockgraphik zu setzen.

Um den Bildschirmspeicher einfacher ansprechen zu können, wenden wir den Trick an, den wir auch schon in Kapitel 2.7 benutzt hatten. Wir nehmen ein Feld mit der Größe von 1024 Bytes (dies entspricht der Größe des Bildschirms) und legen es über den Bildschirmspeicher. Diesen Trick hatten wir schon angewandt, um die Hires-Seite in eine Variable zu fassen. Dadurch können wir einzelne Punkte schneller durch einen Feldindex ansprechen und brauchen nur noch die POKE-Prozedur, um zwischen Graphik- und Textmodus hin und her zu schalten. Unser Feld nennen wir logischerweise "SCREEN".

Bei den Prozeduren halten wir uns einfach an die Basic Befehle, da sie alle nötigen Funktionen erfüllen. Neu hinzu kommt nur die Prozedur "FILL".

Nun zu den Befehlen im einzelnen.

"FILL (COLOR)" färbt die Graphikseite in der Farbe "COLOR" ein. "COLOR" sollte nur von 0 bis 15 gehen. Die sich ergebenden Farben sind beim "PLOT"-Befehl angegeben. Wir benutzen hier einfach die Prozedur "FILLCHAR", die eine Variable mit einem Wert füllt; hier das Feld "SCREEN".

"GR" schaltet den Blockgraphik-Modus ein. Wir POKEen dazu in die oben angegeben Speicherstellen und färben den Bildschirm mit "FILL" schwarz.

"TEXT" schaltet zurück in den Textmodus. Dazu muß die Speicherstelle - 16303 angesprochen werden. Damit der Bildschirm nicht mit unsinnigen Zeichen gefüllt ist, löschen wir ihn.

"COLOR(C)" setzt die Farbe, in der die folgenden Plotbefehle ausgeführt werden sollen. "C" sollte zwischen 0 und 15 liegen, wobei die Werte die folgenden Farben ergeben:

0: schwarz
1: magenta
2: dunkelblau
3: rot
4: dunkelgrün
5: grau
6: blau
7: hellblau
8: braun
9: orange
10: grau
11: rosa
12: grün
13: gelb
14: türkis
15: weiß

Damit wir uns späteren Rechenaufwand sparen, setzen wir gleich die Variablen "COLORN" und "COLORM" auf die Werte, die für den oberen und den unteren Punkt in einer Speicherstelle gebraucht werden.

"PLOT (X,Y)" setzt einen Punkt mit den Koordinaten X und Y in der vorher festgelegten Zeichenfarbe. Wir berechnen mit einer komplizierten Formel die Variable "LOC", die den Index für das Feld

"SCREEN" ergibt. Diese Formel ist deshalb so lang, da der Bildschirmaufbau beim Apple äußerst umständlich ist. Wir schauen noch, um welchen Punkt es sich handelt und setzen das entsprechende Nibble.

"HLIN (X1,X2,Y)" zeichnet eine waagerechte Linie von X1 bis X2 auf der Höhe Y. Es handelt sich um eine einfache Schleife mit dem "PLOT"-Befehl.

"VLIN (Y1,Y2,X)" zeichnet eine senkrechte Linie von Y1 bis Y2 in der Spalte X. Es handelt sich hier wiederum um eine simple Schleife.

"SCRN (X,Y)" ist eine Funktion, die angibt, welche Farbe der Punkt mit den Koordinaten X und Y hat. Das Ergebnis ist eine Zahl zwischen 0 und 15.

Es ist zu beachten, daß nicht geprüft wird, ob die angegeben Koordinaten für die Prozeduren innerhalb des Bildschirms liegen. X Koordinaten dürfen nur von 0 bis 39 und Y-Koordnaten nur von 0 bis 47 gehen.

Nun zum Aufbau der Unit. Auf die im "INTERFACE"-Teil angebenen Typen und Prozeduren kann man von "außen" zugreifen. Dem Programm, das die Unit benutzt, stehen also die Blockgraphik-Prozeduren und der Typ "BYTE" zur Verfügung. Alles, was im "IMPLEMENTATION"-Teil, steht ist intern und kann nicht direkt angesprochen werden. So z.B die Prozedur POKE. Der Teil am Ende der Unit zwischen "BEGIN" und "END." ist der Initialisierungsteil. Er wird beim Programmstart automatisch ausgeführt und legt das "SCREEN"-Feld über den Bildschirmspeicher.

Wichtige Typen und Variablen

"SPCHRINHALT","SPCHRSTELLE" : Aus Kapitel 3.1 von PEEK/POKE übernommen

"SCREENARRAY" : 1024 Bytes großes Feld. Die Größe entspricht der des Bildschirmspeichers

"SCREENTYP" : Mittels des Zeigers "AD" kann eine Variable von diesem Typ über einen bestimmten Speicherbereich gelegt werden.

"COLORN","COLORM" : enthalten den Wert, den nötig ist, um den oberen oder den unteren Punkt in einer Bildschirmzelle in der gewünschten Farbe zu setzen.

"SCREEN" : Variable vom Typ "SCREENTYP". Wird über den Bildschirmspeicher gelegt.

Programm "LORES.LIB"

```
(*$S+*)    (* SWAPPING OPTION FUER UNITS *)
UNIT LORES;


INTERFACE (* ALLE IDENTIFIER, AUF DIE VOM
             HOST-PROGRAMM BENUTZT WERDEN
             KOENNEN *)

TYPE
  BYTE = 0..255;

PROCEDURE FILL (COLOR:BYTE);
PROCEDURE GR;
PROCEDURE TEXT;
PROCEDURE COLOR (C:BYTE);
PROCEDURE PLOT (X,Y:BYTE);
PROCEDURE HLIN (X1,X2,Y:BYTE);
PROCEDURE VLIN (Y1,Y2,X:BYTE);
FUNCTION SCRN (X,Y:BYTE):BYTE;


IMPLEMENTATION (* ALLE IDENTIFIER UND DER TEXT
                  DER UNIT. DIESER TEIL IST
                  "PRIVAT". D.H. ER KANN NICHT
                  VOM HOST-PROGRAMM AUFGERUFEN
                  WERDEN *)

TYPE
  SPCHRINHALT = PACKED ARRAY[0..0] OF BYTE;
  SPCHRSTELLE = RECORD CASE BOOLEAN OF
                   TRUE: (ADRESSE:INTEGER);
                   FALSE:(INHALT :^SPCHRINHALT);
                 END;
  SCREENARRAY = PACKED ARRAY[0..1023] OF BYTE;
  SCREENTYP = RECORD CASE BOOLEAN OF
               TRUE: (AD:INTEGER);
               FALSE:(INH:^SCREENARRAY);
             END;

VAR
  COLORN,COLORM:BYTE;
  SCREEN:SCREENTYP;


PROCEDURE POKE(A:INTEGER; B:BYTE); ( aus Kapitel 3.1 uebernommen )
VAR X:SPCHRSTELLE;
BEGIN
  X.ADRESSE:=A;
  X.INHALT^[0]:=B;
END;

PROCEDURE FILL; ( fuellt Schirm in der Zeichenfarbe )
BEGIN
  FILLCHAR(SCREEN.INH^,1023,(COLOR MOD 16)*17);
END;

PROCEDURE GR;    ( schaltet in den Grafikmodus )
BEGIN
  POKE(-16304,0); (* GRAPHICS *)
```

```
    POKE(-16298,0); (* LORES *)
    FILL(0);
  END;

  PROCEDURE TEXT; { schaltet in den Textmodus }
  BEGIN
    POKE(-16303,0); (* TEXT *)
    WRITE(CHR(12));
  END;

  PROCEDURE COLOR; { setzt Zeichenfarbe }
  BEGIN
    COLORN:=C MOD 16;
    COLORM:=C * 16;
  END;

  PROCEDURE PLOT; { setzt eine Punkt in der Zeichenfarbe }
  VAR LOC,Y1,B:INTEGER;
  BEGIN
    Y1:= Y DIV 2;
    LOC := (Y1 DIV 8) * 40 + (Y1 - 8 * (Y1 DIV 8)) * 128 + X;
    B:=SCREEN.INH^[LOC];
    CASE ODD(Y) OF
      FALSE : B := (B DIV 16) * 16 + COLORN;
      TRUE  : B := (B MOD 16) + COLORM
    END;
    SCREEN.INH^[LOC]:=B;
  END;

  PROCEDURE HLIN; { zieht eine waagerechte Linie in der Zeichenfarbe }
  VAR X:INTEGER;
  BEGIN
    IF X1 <= X2 THEN
      FOR X := X1 TO X2 DO
        PLOT(X,Y)
    ELSE
      FOR X := X1 DOWNTO X2 DO
        PLOT(X,Y);
  END;

  PROCEDURE VLIN; { zieht eine senkrechte Linie in der Zeichenfarbe }
  VAR Y:INTEGER;
  BEGIN
    IF Y1 <= Y2 THEN
      FOR Y := Y1 TO Y2 DO
        PLOT(X,Y)
    ELSE
      FOR Y := Y1 DOWNTO Y2 DO
        PLOT(X,Y);
  END;

  FUNCTION SCRN; { ergibt Farbe eines Punktes }
  VAR LOC,Y1,B:INTEGER;
  BEGIN
    Y1:= Y DIV 2;
    LOC := (Y1 DIV 8) * 40 + (Y1 - 8 * (Y1 DIV 8)) * 128 + X;
    B:=SCREEN.INH^[LOC];
    CASE ODD(Y) OF
      FALSE : SCRN := B MOD 16;
      TRUE  : SCRN := B DIV 16
    END;
```

```
END;

(* INITIALISATION *)
BEGIN
   SCREEN.AD:=1024; (* BEGINN DES BILDSCHIRMSPEICHERS *)
END.
```

Die Unit soll in dieser Form nicht in die "SYSTEM.LIBRARY" eingebunden werden. Jedes Programm, das sie benutzt, muß zu Beginn dem Compiler mitteilen, daß die Unit aus einem anderen File stammt. Dies geschieht mit der "(*$U Filename*)" Option. Darauf folgt die Angabe "USES LORES;".

Nach dem Compileren muß die Unit noch in das Codefile mit dem Linker eingebunden werden. Nach dem Aufruf des Linkers mit "L" von der Hauptkommandozeile ergibt sich der folgende Dialog:

```
LINKING...

APPLE PASCAL LINKER [1.1]
HOST FILE? <ret>
OPENING SYSTEM.WRK.CODE
LIB FILE? LORES.LIB
OPENING LORES.LIB.CODE
LIB FILE? <ret>
MAP FILE? <ret>
READING LORESDEM
READING LORES
OUTPUT FILE? <ret>
LINKING LORES    # 7
LINKING LORESDEM # 1
```

Bild 4.3.2: Der Dialog beim Linkerlauf

Der Linker muß hier übrigens einzeln aufgerufen werden. Falls er über das "R" (Run) Kommando nach dem Compilieren gestartet wird, ist es nicht möglich, die Unit einzubinden. Dies liegt daran, daß der Linker dann annimmt, daß die Unit in der "SYSTEM.LIBRARY" steht, wo er sie natürlich dann nicht findet.

Um die Anwendung zu demonstrieren ist hier noch ein kurzes Beispielprogramm abgedruckt, das einige Blockgraphik- Prozeduren benutzt. Es läßt einen kleinen Ball auf dem Bildschirm hin und herfliegen. Im Aufbau entspricht es dem Basic-Beispielprogramm auf Seite 10 des "Applesoft Basic Programming Reference Manual".

Programm "LORES.DMO"

```
PROGRAM LORESDEMO;

(*$U LORES.LIB.CODE *) (* NAME DES UNIT-FILES *)
USES LORES;  (* LIBRARY LORES WIRD BENUTZT *)

VAR
  I,X,Y,NX,NY,XV,YV:INTEGER;

BEGIN
```

```
   GR;
   COLOR(10);
   HLIN(0,39,0);
   VLIN(0,47,39);
   HLIN(39,0,47);
   VLIN(47,0,0);
   I:=0;
   X:=1; Y:=5;
   XV:=2; YV:=1;
   REPEAT
     NX:=X+XV;
     NY:=Y+YV;
     IF NX > 38 THEN BEGIN
                       NX:=38; XV:=-XV;
                     END;
     IF NX < 1 THEN BEGIN
                      NX:=1; XV:=-XV;
                    END;
     IF NY > 46 THEN BEGIN
                       NY:=46; YV:=-YV;
                     END;
     IF NY < 1 THEN BEGIN
                      NY:=1; YV:=-YV;
                    END;
     COLOR (13);
     PLOT (NX,NY);
     COLOR (0);
     PLOT (X,Y);
     X:=NX; Y:=NY;
     I:=I+1;
   UNTIL I > 1000;
   READLN;
   TEXT;
END.
```

4.4 Shapes in der Lores-Graphik

In diesem Kapitel soll noch eine kurze Erweiterung der Blockgraphik-Möglichkeiten besprochen werden. Es handelt sich um die Darstellung von Shapes auf dem Bildschirm. Größere Objekte brauchen damit nicht mehr Punkt für Punkt in die Graphik gezeichnet werden. Sie entpricht der "DRAWBLOCK" Prozedur aus der "Turtlegraphics" Unit.

Dazu ist es nötig, das zu zeichnende Objekt in ein Feld zu fassen. Die Größe des Feldes wird durch die Höhe und die Breite der Figur bestimmt. Da es in Pascal nicht möglich ist, Felder mit variabler Größe an eine Prozedur zu übergeben, muß die Prozedur auf eine bestimmte Größe festgelegt werden.

Jedes Element in dem Feld gibt einen Punkt wieder. Der Feldindex bestimmt seine Position auf dem Bildschirm. Da ein Punkt 16 Farben annehmen kann, hat jedes Feldelement den Wertebereich 0 bis 15.

Ein Prozedur, die das Objekt auf den Bildschirm bringt ist einfach. Das o.g. Feld wird Zeile für Zeile durchgegangen und die Punkte in der entsprechenden Farbe mit dem "PLOT"-Befehl ausgegeben. Dieser Vorgang ist leider nicht schnell genug, um Figuren in einer annehmbaren Zeit zu bewegen. Sie vereinfachen aber die Ausgabe von farbigen Symbolen auf dem Bildschirm ungemein.

Das hier abgedruckte Programm bringt einen farbigen Ball fünfzigmal an zufälligen Positionen auf den Bildschirm und untermalt das Ganze mit Tönen.

Damit das Programm läuft, ist wieder ein Lauf des Linkers nötig. Es ergibt sich dabei der folgende Dialog:

```
LINKING...


APPLE PASCAL LINKER [1.1]
HOST FILE? <ret>
OPENING SYSTEM.WRK.CODE
LIB FILE? LORES.LIB
OPENING LORES.LIB.CODE
LIB FILE? <ret>
MAP FILE? <ret>
READING LORESSHA
READING LORES
OUTPUT FILE? <ret>
LINKING LORES    # 7.
LINKING LORESSHA # 1
```

Bild 4.4.1: Der Dialog beim Linkerlauf

Ansonsten gilt auch hier das in Kapitel 4.3 zum Linkerlauf Gesagte.

Wichtige Typen, Variablen und Prozeduren

"SHAPEHOEHE" : Konstante, die die Höhe des Shapes angibt.

"SHAPEBREITE" : Konstante, die die Breite des Shapes angibt. Soll mit einer anderen Shapegröße gearbeitet werden, so müssen nur diese zwei Konstanten geändert werden.

"SHAPETYP" : Feld, das die Definition des Objekts enthalten soll.

"SHAPE" : Variable vom Typ "SHAPETYP". Enthält das Aussehen des Balls.

"DRAWSHAPE (X,Y:INTEGER)" : Prozedur, die das Feld "SHAPE" plottet. X und Y geben die Koordinaten der linken oberen Ecke der Figur an.

Programm "LORES.SHP"

```
{$S+}
{$R-} { Rangechecking ausschalten, um Programm schneller zu machen }

program lores_shapes;

uses applestuff, {$U LORES.LIB.CODE} lores;         { Unit benuzten }

const shapehoehe = 5;
      shapebreite = 5;

type shapetyp = array [1..shapehoehe,1..shapebreite] of 0..15;

var shape: shapetyp;
    i,x,y: integer;

procedure initshape; { definiert Aussehen des Shapes }
var i:integer;
begin
  shape[1,1]:=0; shape[1,2]:=0; shape[1,3]:=3; shape[1,4]:=0; shape[1,5]:=0;
  shape[2,1]:=0; shape[2,2]:=4; shape[2,3]:=4; shape[2,4]:=4; shape[2,5]:=0;
  for i:= 1 to 5 do
    shape[3,i]:=6;
  shape[4,1]:=0; shape[4,2]:=1; shape[4,3]:=1; shape[4,4]:=1; shape[4,5]:=0;
  shape[5,1]:=0; shape[5,2]:=0; shape[5,3]:=8; shape[5,4]:=0; shape[5,5]:=0;
end;

procedure drawshape (x,y:integer); { zeichnet Shape }
var i,j:integer;
begin
  for i:= 1 to shapehoehe do
    for j:= 1 to shapebreite do
      begin
        color(shape[i,j]);
        plot(x+j-1,y+i-1);
      end;
end;

begin
  randomize;
  initshape;
  gr;
  i:=0;
  repeat
    x:=random mod 36;
    y:=random mod 44;
    drawshape(x,y);
    note(random mod 51,10);
    i:=i+1;
  until i>50;
  readln;
  text;
end.
```

4.5 Editor-Marken verändern und löschen

Man kann mit dem Pascal-Editor Marken ("markers") setzen und ihnen einen Namen geben. Sie können dann als Sprungziele innerhalb eines Textes benutzt werden. Man setzt sie mit "S M". Was ist aber, wenn man eine Marke nicht mehr braucht, oder sich verschrieben hat ? Nach dem Drücken von <ret> bei der Markeneingabe versagt der Editor. Man kann Marken nur dann löschen oder ändern, wenn man schon zehn Marken gesetzt hat, und Platz für eine neue machen muß.

Aus Kapitel 2.8 wissen wir, daß alle Informationen, die der Editor braucht (also auch die Namen und Positionen der Marken) in den ersten zwei Blöcken eines jeden Textfiles abgespeichert werden. Mit diesem Wissen ist es einfach, ein Programm zu schreiben, das diese Informationen liest, anzeigt und verändern kann. Unser Utility wird dann nach dem Verlassen des Editors aufgerufen und arbeitet mit einem Textfile.

Man hat einige Punkte beim Verändern der Marken zu beachten. Zunächst müssen die Marken aufeinander folgen. Es kann also nicht z.B. die erste und zweite Marke benutzt und die dritte unbenutzt sein, wenn noch die vierte benutzt wird. Eine unbenutzte Marke wird gekennzeichnet, indem das erste Zeichen ein CHR(0) ist.

Nun also zum Programm selbst. Zuerst wird mittels der Struktur aus Kapitel 2.8 die gewünschten Informationen für ein Textfile eingelesen. Werte wie rechter, linker Rand etc. werden am Bildschirm angezeigt, wobei das Layout dem des Editors entspricht. Dann folgen die zehn Marken. Unbenutzte Marken werden durch "UNUSED" gekennzeichnet.

Der Benutzer hat nun die Möglichkeit, eine Marke zu verändern. Durch Auswahl über die Ziffern 0 bis 9 kann er einen neuen Namen für die Marke eingeben, der maximal 8 Zeichen lang sein kann und durch <ret> abgeschlossen wird. Gibt er nur <ret> ein, so wird die Marke gelöscht. Alle nachfolgenden Marken werden aufgrund der obigen Überlegungen um einen Platz nach vorne geschoben. Gibt er <esc> <ret> ein, so bleibt alles beim alten. Logischerweise wird das Verändern von unbenutzten Marken verweigert.

Mit "Q" wird das Programm verlassen. Der erste Block des Textfiles wird mit den neuen Editor-Informationen überschrieben, und wir können den Text nun mit den veränderten Marken editieren.

Achtung: Falls der Leser nicht mit dem normalen Pascal-Editor arbeiten sollte, kann es sein, das sein Editor die ersten zwei Blöcke eines Textfiles für andere Informationen benutzt. "MARKERCHANGE" arbeitet dann wahrscheinlich nicht korrekt.

Wichtige Variablen

"TEXTINFO" : Strukturierter Record zur Aufnahme der Informationen am Beginn eines Textfiles

Wichtige Prozeduren

"LESEINFO" : Fragt nach dem Namen des Textfiles und liest mittels der Prozedur "BLOCKREAD" die Variable "TEXTINFO" ein. Bei einem Fehler wird "LESEINFO" wiederholt.

"GEBEAUS" : Gibt die Variable "TEXTINFO" aus. Das Schirmlayout entspricht dem des Editors. Im Anschluß folgen die 10 Marken.

"VERAENDERE" : Liest Benutzerkommandos ein und führt sie aus, bis ein "Q" zum Beenden eingegeben wird.

"SCHREIBEINFO" : Schreibt textinfo in den ersten Block des Textfiles zurück.

Kommandos:

"0" - "9" : Marke auswählen
dann <esc> <ret> : Alles beim alten lassen
<ret> : Marke löschen
Texteingabe : Neuer Name der Marke

"Q" : Beenden des Programms (Quit)

Programm "MARKERS.TEXT"

```
program markerchange;

type datum = packed record
               monat : 0..12;
               tag   : 0..31;
               jahr  : 0..99;
             end;

var textinfo: packed record
                unused1          : packed array[0..3] of char;
                markername       : packed array[0..9,0..7] of char;
                unused2          : packed array[0..9] of char;
                markeradresse    : packed array[0..9] of integer;
                autoindent       : integer;
                filling          : integer;
                tokendef         : integer;
                leftmargin       : integer;
                rightmargin      : integer;
                paramargin       : integer;
                commandch        : char;
                datecreated      : datum;
                lastused         : datum;
                unused3          : packed array[0..379] of char;
              end;
    txtfile : file;
    home    : char;
    esc     : string[1];
    filename: string;

procedure leseinfo; { liest Textinformationen ein }
var    ok: boolean;
    dummy: integer;
begin
  write(home);
  writeln('MARKERCHANGE');
  writeln('------------');
  writeln('Veraendern und loeschen von Markern');
  writeln('in Editor-Texten.');
  writeln;
  repeat
    write('Filename (ohne .TEXT) : ');
```

```
      readln(filename);
      filename:=concat(filename,'.TEXT');
      ok:=true;
      {$I-}
      reset(txtfile,filename);
      {$I+}
      if ioresult<>0 then begin
                            writeln('Kann ''',filename,''' nicht oeffnen');
                            writeln('Bitte nochmals .');
                            ok:=false;
                          end;
   until ok;
   dummy:=blockread(txtfile,textinfo,1,0);
   close(txtfile);
end;

procedure gebeaus; { gibt Textinformationen aus }
var i:integer;
begin
  write(home);
  with textinfo do
    begin
      write('Auto indent  ');
      if autoindent>0 then writeln('True')
                      else writeln('False');
      write('Filling      ');
      if filling   >0 then writeln('True')
                      else writeln('False');
      writeln('Left margin  ',leftmargin);
      writeln('Right margin ',rightmargin);
      writeln('Para margin  ',paramargin);
      writeln('Command ch   ',commandch);
      write('Token def    ');
      if tokendef  >0 then writeln('True')
                      else writeln('False');
      writeln;
      with datecreated do
        writeln('Date Created ',monat,'-',tag,'-',jahr);
      with lastused do
        writeln('Last Used    ',monat,'-',tag,'-',jahr);
      writeln;
      for i:=0 to 9 do
        begin
          write('Marker No. ',i,' : ');
          if markername[i,0] = chr(0) then writeln('  Unused')
                                      else writeln('>',markername[i],'<');
        end;
    end;
end;

procedure veraendere; { laesst der Benutzer Marker aendern oder loeschen }
var   ch:char;
    i,no:integer;
 newname:string;
begin
  repeat
    gotoxy(0,22);
    write('<0..9> Marker aendern  <Q> Beenden ',chr(8));
    read(ch);
    if ch in ['0'..'9'] then
      with textinfo do
```

```
          begin
            no:=ord(ch)-48;
            if markername[no,0]=chr(0) then
              begin
                gotoxy(0,22);
                write('Marker ist unbenutzt. Bitte Taste. ',chr(8));
                read(ch);
              end
            else
              begin
                gotoxy(0,22);
                write('<esc><ret> zurueck   <ret> loeschen');
                gotoxy(16,11+no);
                write('          ');
                gotoxy(16,11+no);
                readln(newname);
                if newname=''
                  then begin
                         for i:=no to 8 do
                           begin
                             markername[i]:=markername[i+1];
                             gotoxy(15,11+i);
                             if markername[i,0] = chr(0)
                               then writeln('  Unused  ')
                               else writeln('>',markername[i],'<');
                           end;
                         markername[9,0] := chr(0);
                         gotoxy(15,20);
                         writeln('  Unused  ')
                       end
                  else if newname<>esc
                    then begin
                           newname:=concat(newname,'          ');
                           for i:= 0 to 7 do
                             markername[no,i]:=newname[i+1];
                         end;
                gotoxy(15,11+no);
                if markername[no,0] = chr(0)
                  then writeln('  Unused  ')
                  else writeln('>',markername[no],'<');
              end;
          end;
  until ch in ['Q','q'];
  write(chr(8),' Quit');
end;

procedure schreibeinfo; { schreibt Textinformationen zurueck auf die Diskette }
var dummy:integer;
begin
  reset(txtfile,filename);
  dummy:=blockwrite(txtfile,textinfo,1,0);
  close(txtfile,lock);
end;

begin
  esc:=' ';
  esc[1]:=chr(27);
  home:=chr(12);
  leseinfo;
  gebeaus;
  veraendern;
```

```
  schreibeinfo;
end.
```

4.6 Wie groß ist mein Text ?

Neben dem Inhalt und dem Aussehen eines Textes ist noch seine Größe ein entscheidendes Merkmal. Wer einen Artikel, ein Buch oder sonst ein Schriftstück erstellt, der benötigt diese Information, um abzuschätzen, ob er Anforderungen an den Umfang seines Textes erfüllt hat, oder z.B., wieviele Seiten der Text auf dem Drucker ergeben wird. Bei vielen Texteditoren gibt es die Möglichkeit, sich diese Werte ausgeben zu lassen; beim Pascal-Editor nicht. Man kann nur feststellen, wieviel Speicherplatz der Text benötigt und wieviel Diskettenplatz.

Die besten Informationen über die Größe eines Textes bieten wohl die Angaben, wieviele Zeichen, Wörter und Zeilen der Text enthält. Aus der Zeilenzahl läßt sich dann noch errechnen, wieviele Seiten der Text ergibt. Das folgende Programm errechnet diese Werte und gibt sie aus.

Die Realisation ist relativ einfach. Wir benötigen zunächst den Namen des Textfiles, das bearbeitet werden soll. Dann müssen wir wissen, was gezählt werden soll. Falls man nur die Zeilenzahl wissen will ist es nicht nötig, die Wörter zu zählen, wobei dies ja auch den Programmablauf verlängern würde. Schließlich wird noch ein Wert benötigt, der angibt, wieviele Zeilen auf eine Druckseite kommen, um die Seitenanzahl durch Division zu berechnen.

Das Programm öffnet nun das Textfile und liest den Text Zeile für Zeile. Daraus ergibt sich automatisch die Anzahl der Zeilen, da ja nur bei jedem Lesevorgang ein Zähler um 1 erhöht werden braucht.

Auch das Zählen der Zeichenanzahl ist einfach. Sie ergibt sich aus den Summe der Länge der Zeilen. Hierbei werden natürlich Leerstellen mitgezählt, was allerdings unerheblich ist. Falls die Wörter gezählt werden, ist es möglich, auch noch diesen kleinen Fehler zu beheben. Wenn man pro Wort einen Leerraum rechnet, so kommt man der wirklichen Zeichenzahl ohne Leerstellen durch die Berechnung "Zeichenzahl minus Anzahl der Wörter" ziemlich nahe. Unser Programm führt diese Berechnung automatisch durch, falls Zeichen und Wörter gezählt werden.

Am "aufwendigsten" ist das Zählen der Wörter. Ein Wort wird definiert als eine Zeichenfolge, die von Leerstellen begrenzt ist. Das Programm nimmt also die gerade eingelesene Zeile, und sucht das erste Zeichen, das keine Leerstelle ist. Mit diesem Zeichen beginnt ein Wort, und wir können den Wortzähler um 1 erhöhen. Dann wird das nächste Leerzeichen gesucht, das ja ein Wort beendet. Da auch mehrere Leerstellen zwischen Wörtern stehen können, beginnt nun wieder die Suche nach einem Zeichen, das keine Leerstelle ist. Dies wiederholt sich bis zum Ende der Zeile.

Diese Methode ist zwar sehr einfach, sie wird aber nicht immer die korrekte Anzahl der Wörter ergeben. Wenn nämlich ein Wort an Ende einer Zeile getrennt wurde, so wird es zweimal gezählt. Einmal am Ende der einen Zeile und einmal am Anfang der nächsten. Da aber dieser Fehler nur eine geringe Abweichung ergibt, und nie die genaue Wörteranzahl gebraucht wird, können wir ihn in Kauf nehmen und brauchen das Programm nicht unnötig zu vergrößern und zu verlangsamen.

Falls die Seitenanzahl benötigt wurde, so wird sie aus den mitgezählten Zeilen mittels der oben genannte Division ermittelt und ausgegeben.
Schließlich bauen wir in das Programm noch eine optische Kontrolle ein, damit erkennbar ist, daß daß Programm am Arbeiten ist. Jedesmal, wenn eine Zeile eingelesen wird, wird auf dem Bildschirm ein Stern ("*") ausgegeben. Sind fünf Zeilen eingelesen, so werden die Sterne gelöscht, und der Vorgang beginnt neu. Damit ergibt sich ein "Pulsieren" der Anzeige.

Es ergibt sich das folgende Listing, wobei die Prozeduren "FRESET","FREADLN" und "FCLOSE" aus Kapitel 2.9 übernommen wurde. Ohne sie benötigt das Programm weit mehr Zeit; der Leser kann es ausprobieren, indem er die Standardprozeduren verwendet.

Wir hatten aber schon in Kapitel 2.9 bemerkt, daß die dort abgedrukkten Prozeduren nur auf einen Text mit bekannter Länge angewandt werden können. Sie hatten den Nachteil, daß das Ende des Textes nicht erkannt wurde. Beim Zählen eines beliebigen Textes ist seine Größe natürlich nicht vorher bekannt, um wir müssen die Prozeduren etwas erweitern.

Wir führen zwei neue Werte ein, die mit einem Textfile zusammenhängen. Es handelt sich um die Anzahl der Blöcke, die das File auf der Diskette belegt, und die Anzahl der gültigen Zeichen im letzten Block. Wie wir diese Informationen erhalten wird weiter unten beschrieben.

Mit diesen zwei Werten führen wir Änderungen an "FREADLN" durch. Wir wollen erreichen, daß das Ende eines Textes erkannt wird. Damit nach dem Aufruf der Prozedur bekannt ist, ob schon das Ende erreicht ist, wird an "FREADLN" die Boolean-Variable "EF" übergeben. Sie hat die gleiche Funktion wie die "EOF"-Variable im Zusammenhang mit dem Standard-"READLN". Wenn bereits alle Textzeilen eingelesen sind, wird sie auf "TRUE" gesetzt. Hier ist zu bemerken, daß der dann gelesene String ungültig ist.

Das Ende des Textes wird auf zwei Arten erkannt. Zunächst, wenn ein neuer Block in den Puffer "SEITE" geladen werden soll. Durch Vergleichen mit dem Wert, der angibt, wieviele Blöcke das File groß ist, wird festgestellt, ob schon der letzte Block eingelesen war. In diesem Fall ist das Ende des Files erreicht.

Ist der Block, der sich in "SEITE" befindet der letzte Block im File, so wird geprüft, ob der Seitenzeiger schon über das Ende des Textes hinauszeigt. Dies ist der Fall, wenn er größer ist als, der Wert, der angibt, wieviele Bytes im letzten Block gültige Textzeichen sind.

Der Text wird also zeilenweise mit "FREADLN" eingelesen, solange bis "EF" "TRUE" wird. Dann ist das Programm fertig und kann die Ergebnisse ausgeben.

Nun muß noch beschrieben werden, wie wir die Wert für die Anzahl der Blöcke und der gültigen Zeichen im letzten Block erhalten. Dazu ist das Wissen aus Kapitel 2.2 über das Directory nötig.

Für jedes File sind im Directory die zwei Werte vermerkt, die wir benötigen. Die Anzahl der Blöcke ergibt sich aus der Differenz zwischen der Nummer des Anfangs- und des Schlußblocks des Files. Der zweite Wert steht direkt im Directory.

Wir brauchen also nur das entsprechende Directory einlesen, und die zwei Werte setzten. Wir wissen allerdings nicht, von welchem Laufwerk wir das Directory lesen müssen. Dies hängt direkt von dem Filenamen ab, den der Benutzer eingibt.

Es bleibt uns nichts anderes übrig, als die Diskette zu suchen. Wir berücksichtigen dabei die Laufwerke 4 und 5.

Zunächst wird der eingegebene Filename untersucht. Ist sein erstes Zeichen ein "#", so muß darauf eine Unit-Nummer folgen. Ist diese 4 oder 5, so haben wir das gesuchte Laufwerk schon gefunden. Kommt eine nicht erlaubte Unit-Nummer vor, so wird das Programm mit einer Fehlermeldung abgebrochen.

Weiterhin kann es sein, daß ein Volumename angegeben worden ist. Wir lesen nun die Directorys aus beiden Laufwerken ein (zunächst von Unit 4, dann eventuell von Unit 5), und vergleichen den darin vorkommenden Volume-Namen mit dem eingegebenen. Stimmen sie überein, so haben wir das Laufwerk gefunden. Wird der angegebene Volume-Name nicht gefunden, so bricht das Programm wieder mit einer Fehlermeldung ab.

Es gibt noch den Fall, daß gar kein Volume-Name angegeben worden ist. Nun wird vor den eingebenen Filename der Volume-Name der Prefix-Diskette gesetzt. Wir erhalten ihn über eine an "PEEK"/"POKE" angelehte Vorgehensweise, die in Kapitel 3.3 näher erläutert ist. Daraufhin gehen wir wie oben vor.

Wir wissen zu diesem Zeitpunkt, in welchem Laufwerk (= auf welcher Unit) sich der zu zählende Text befinden müßte. Ob er dort auch wirklich ist, prüfen wir nach, indem wir das Directory nach dem Filenamen durchsuchen. Finden wir den Text, so können wir die zwei für "FREADLN" benötigten Werte setzen, andernfalls wird das Programm mit einer Fehlermeldung verlassen.

Wir unschwer zu erkennen ist, ist dies ein recht komplexer Aufwand. Bei der Verwendung der Standard-Prozedur "READLN" wird er vom Betriebssystem übernommen. Da wir hier nicht direkt eingreifen können, müssen wir diesen Teil neu schreiben. Im Grunde genommen ist er auch noch nicht vollständig korrekt. So fehlt z.B. die Erkennung des "Root"-Volumes ("*") und eine Behandlung aller möglichen Fehleingaben. Dieser Aufwand ist aber durch den enormen Geschwindigkeitsvorteil von "FREADLN" sich gerechtfertigt.

Programm "COUNIT.TEXT"

```
program countit;

type
  vname=string[7];  { Ein Volumename }
  fname=string[15]; { Ein Filename }
  daterec = packed record { das Datum }
             month: 0..12;
             day:   0..31;
             year:  0..100;
            end;
  fkind = (vol,bad,code,text,info,data,graf,foto,secr); { moegliche Filetypen }
  direntry = record { ein Eintrag im Directory }
              firstblock: integer; { Block, bei dem das File anfaengt }
              lastblock: integer;  { Block, bei dem das File endet }
```

```
                case filekind: fkind of
                     { nur beim Eintrag #0 }
                     vol,secr: (diskvol: vname;    { Name der Diskette }
                                blockno: integer; { Anzahl der Bloecke }
                                fileno:integer;    { Anzahl der Files }
                                accesstime: integer; { unbenutzt }
                                lastboot: daterec);  { Datum, an dem die     }
                                                     { zuletzt benutzt wurde }
                     { Normale File-Eintraege }
                     bad,code,text,
                     info,data,graf,
                     foto: (filename: fname; { Filename }
                            endbytes: 1..512; { Bytes im letzten Block }
                            lastaccess: daterec); { Datu, an dem das File }
               end;                                  { geschrieben wurde }
  direc = array[0..77] of direntry; { Directory mit 78 Eintraegen }

var
{ Variablen, die von "FRESET","FREADLN" und "FCLOSE" benutzt werden }
             f:file;
         seite:packed array[0..1023] of char;
  bufferpointer:integer;
   blockpointer:integer;
           leer:string[120];

{ Variablen, zum Festlegen des Fileendes }
   blockno,lastbytes:integer;
   filename:string;

{ Variablen zum Zaehlen }
  zeilen,zeichen,woerter,seiten:boolean;
  zeilenno,zeichenno,wortno:integer;
  seitenzeilen:integer;

procedure inputoptions; { Eingeben, was gezaehlt werden soll }
var ch:char;
begin
  writeln('---------------------------------------');
  writeln('                    COUNT');
  writeln('---------------------------------------');
  writeln;
  write('Zeilen zaehlen >');
  read(ch);
  if ch in ['N','n'] then zeilen:=false
                     else zeilen:=true;
  writeln;
  writeln;
  write('Zeichen zaehlen >');
  read(ch);
  if ch in ['N','n'] then zeichen:=false
                     else zeichen:=true;
  writeln;
  writeln;
  write('Woerter zaehlen >');
  read(ch);
  if ch in ['N','n'] then woerter:=false
                     else woerter:=true;
  writeln;
  writeln;
  write('Seiten errechnen >');
  read(ch);
```

```
   if ch in ['N','n'] then seiten:=false
                      else seiten:=true;
   writeln;
   writeln;
   if seiten then
     begin
       write('Zeilen pro Seite >');
       readln(seitenzeilen);
     end;
   writeln;
end;

procedure inputname; { Filenamen eingeben }
var prefix: record case boolean of
              true: (adresse:integer);
              false: (name:^vname)
            end;
         i:integer;
begin
  write('Welcher Text >');
  readln(filename);
  if (pos(':',filename)=0) and (filename[1]<>'#') then
    begin { Falls keine Volume-Angabe, dann Prefix einsetzen }
      prefix.adresse:=-22008;
      filename:=concat(prefix.name^,':',filename);
    end;
  for i:=1 to length(filename) do { In Grossbuchstaben umwandeln }
    if ord(filename[i])>95 then
      filename[i]:=chr(ord(filename[i])-32);
  writeln;
end;

procedure findefile; { Suchen, in welchem Laufwerk sich das File befindet }
var  filen:string;
{ Variablen zum Einlesen des Directorys }
 directory:direc;
    unitno:integer;

  procedure lese_dir; { liest das Directory von der Unit #4 ein }
  var io:integer;      { Bei einem Disketten-Fehler wird das Programm verlassen }
    begin
      {$I-}
      unitread(unitno,directory,sizeof(directory),2);
      {$I+}
      io:=ioresult;
      if io<>0 then
        { Fehler ! }
        begin
          writeln(chr(7)); { Beep }
          writeln('Fehler #',io);
          writeln('Directory von Unit #',unitno,' nicht zu lesen');
          exit(program)
        end;
    end;

  procedure findevolume; { Sucht Laufwerk indem der Volumename der     }
  var volume:string;      { Diskette mit dem angebenen verglichen wird }
    begin
      if filename[1]='#'
        then
          begin { Unit-Angabe anstatt Volume }
```

```
            if pos(':',filename)<>3 then { Nur Units 4 & 5 erlaubt }
              begin
                writeln('Fehler> Falsche Unitangabe');
                exit(program);
              end;
            unitno:=ord(filename[2])-48;
            if not (unitno in [4,5]) then { Nur Units 4 & 5 erlaubt }
              begin
                writeln('Fehler> Falsche Unitangabe');
              end;
            lese_dir;  { Directory einlesen }
          end
        else
          begin
            volume:=copy(filename,1,pos(':',filename)-1); { gesuchtes Volume }
            unitno:=4;  { Zunaechst auf Unit 4 nachschauen }
            lese_dir;
            if directory[0].diskvol<>volume then
              begin
                unitno:=5; { Dann auf Unit 5 nachschauen }
                lese_dir;
                  if directory[0].diskvol<>volume then
                    begin  { Nicht gefunden }
                      writeln('Fehler: Volume nicht vorhanden');
                      exit(program);
                    end;
              end;
          end;
    end;

  procedure findeentry; { Festellen, in welchem Directoryeintrag }
  var filen:string;     { sich die Angaben zu dem File befinden  }
          i:integer;
  begin
    filen:=copy(filename,pos(':',filename)+1,
                length(filename)-pos(':',filename));
    i:=0;
    repeat
      i:=i+1;
    until (directory[i].filename=filen) or (i>directory[0].fileno);
    if i>directory[0].fileno then
      begin
        writeln('Fehler> File nicht vorhanden');
        exit(program);
      end;
    { Ende des Textes feststellen }
    blockno:=directory[i].lastblock-directory[i].firstblock;
    lastbytes:=directory[i].endbytes;
  end;

begin
  findevolume;
  findeentry;
end;

procedure freset(n:string); { Oeffnet das File mit dem Namen 'N' }
var dummy:integer;
begin
  reset(f,n);
  dummy:=blockread(f,seite,2,2); { Erste Seite einlesen }
  bufferpointer:=0;              { Zeichenzeiger auf das erste Zeichen setzen }
```

```
  blockpointer:=4;              { Blockzeiger auf die naechste Seite im }
end;                            { File setzen }
{ Den String 'S' aus dem File lesen }
{ 'ef' wird 'true', wenn das Ende des Textes erreicht ist }
procedure freadln(var s:string; var ef:boolean);
var  n,no,start2,dummy:integer;

begin
  s:='';
  ef:=false;
  if (seite[bufferpointer]=chr(0)) or
     (bufferpointer=1024) then    { Eventuell naechste Seite einlesen }
    begin
      if blockpointer>=blockno then { Fileende erreicht }
        begin
          ef:=true;
          exit(freadln);
        end;
      dummy:=blockread(f,seite,2,blockpointer);
      blockpointer:=blockpointer+2;
      bufferpointer:=0;
    end;
  if (blockpointer>=blockno) and (bufferpointer>= lastbytes) then
    begin  { Textende erreicht }
      ef:=true;
      exit(freadln);
    end;
  if seite[bufferpointer]=chr(16) { DLE !}
    then begin
           bufferpointer:=bufferpointer+1;
           n:=ord(seite[bufferpointer])-32; { Anzahl der Blanks holen }
           bufferpointer:=bufferpointer+1;
           if n>0 then s:=copy(leer,1,n); { Blank an den String setzen }
         end;
  no:=scan(1024,=chr(13),seite[bufferpointer]); { Nach CR suchen }
  if no>0 then
    begin
      start2:=length(s);
      s:=concat(s,copy(leer,1,no));   { String zunaechst mit Blanks fuellen  }
      moveleft(seite[bufferpointer],s[start2+1],no); { Tatsaechliche Zeichen }
      bufferpointer:=bufferpointer+no+1;       { in den String bringen, und   }
    end                                        { Zeichenzeiger korrigieren     }
    else bufferpointer:=bufferpointer+1;
end;

procedure fclose; { File schliessen }
begin
  close(f);
end;

procedure count; { Zaehlt Zeilen, Zeichen und Woerter }
var              s:string[120];
          endoffile:boolean;
  zaehler,zeiger,i:integer;
begin
  { Werte auf 0 setzen }
  zeilenno:=0;
  zeichenno:=0;
  wortno:=0;
  zaehler:=0;
  repeat
```

```
      freadln(s,endoffile);
      { Bildschirmanzeige }
      if zaehler=5
        then begin
               for i:=1 to 5 do
                 write(chr(8),' ',chr(8));
               zaehler:=0;
             end
        else begin
               write('*');
               zaehler:=zaehler+1;
             end;
      if not endoffile then
        begin
          zeilenno:=zeilenno+1; { Zeile gezaehlt }
          if zeichen then
            zeichenno:=zeichenno+length(s);  { Zeichen gezaehlt }
          if woerter then
            if scan(length(s),<>' ',s) <> length(s) then
              begin
                zeiger:=1;
                repeat
                  { Zeiger auf Wortanfang setzen }
                  zeiger:=zeiger+scan(length(s)-zeiger,<>' ',s[zeiger]);
                  { Wort zaehlen }
                  if zeiger<=length(s) then
                    wortno:=wortno+1;
                  { Zeiger auf Wortende setzen   }
                  zeiger:=zeiger+scan(length(s)-zeiger,=' ',s[zeiger]);
                until zeiger>=length(s);
              end;
        end;
  until endoffile;
  { Bildschirmanzeige korrigieren }
  if zeiger>0 then
    for i:= 1 to zaehler do
      write(chr(8),' ',chr(8));
  writeln;
end;

procedure giveout; { Ergebnisse ausgeben }
begin
  writeln(chr(12));
  writeln('----------------------------------------');
  writeln('Ergebnis fuer ',filename);
  writeln('----------------------------------------');
  writeln;
  writeln('Zeilen  : ',zeilenno);
  writeln;
  if zeichen then
    begin
      write('Zeichen : ',zeichenno);
      if woerter then writeln('  Korrigiert : ',zeichenno-wortno)
                 else writeln;
    end;
  writeln;
  if woerter then
    begin
      writeln('Woerter : ',wortno);
      writeln;
    end;
```

```
  if seiten then
    writeln('Seiten  : ',(zeilenno div seitenzeilen)+1);
end;

begin
  { 'leer' initialisiern  ( 'leer' kann nicht als }
  { Konstante definiert werden)                   }
  leer:='                                         ';
  leer:=concat(leer,leer,leer);
  { Eingaben }
  inputoptions;
  if (zeilen=false) and (zeichen=false) and
    (woerter=false) and (seiten=false) then
      exit(program); { Falls alle Fragen mit 'n' beantwortet wurden }
  inputname;
  { File suchen }
  findefile;
  { File oeffnen und zaehlen }
  freset(filename);
  count;
  fclose;
  { Ergebnisse ausgeben }
  giveout;
end.
```

Die Benutzung des Programmes ergibt sich von selbst. Man startet es, gibt die erwarteten Eingaben und kann dann die Werte ablesen. Falls man das Programm aus Versehen gestartet hat, so gibt man einfach viermal hintereinander ein "n" ein, und das Programm wird abgebrochen.

Da nach dem Programmlauf wieder die Kommandozeile von POS erscheint, sollte man sofort die Werte ablesen. Ein <ret> danach z.B. löscht bekanntlich den Bildschirm und damit auch die Informationen.

Wichtige Variablen und Prozeduren

"ZEILEN","ZEICHEN",
"WOERTER","SEITEN" : Boolean-Variablen, die angeben, ob Zeilen, Zeichen etc. gezählt werden sollen

"ZEILENNO",
"ZEICHENNO","WORTNO" : Zähler für Zeilen, Zeichen und Wörter

"SEITENZEILEN" : Gibt an, wieviele Zeilen sich auf einer Druckerseite befinden werden

"INPUTOPTIONS" : Fragt den Benutzer, was gezählt werden soll

"INPUTNAME" : Fragt nach dem Namen des zu zählenden Files, und hängt eventuell den Volumenamen des Prefix an

"FINDEFILE" : Sucht das File auf beiden Laufwerken und stellt fest, wie groß das File ist, und wieviele Zeichen im letzten Block gültig sind

"FRESET","FREADLN",

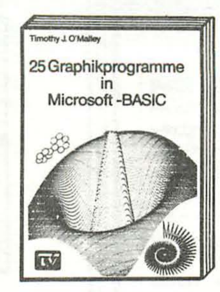
"FCLOSE"	: Aus Kapitel 2.9 übernommen und modifiziert
"COUNT"	: Zählt die gewünschten Werte für das File
"GIVEOUT"	: Gibt die Ergebnisse auf dem Bildschirm aus

Software & Computer-Bücher







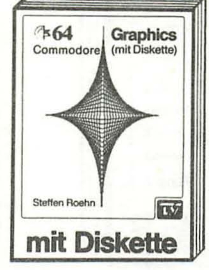


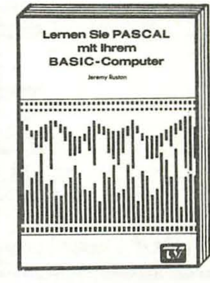




























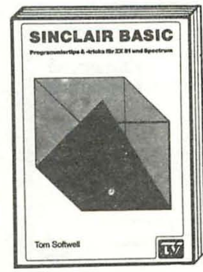














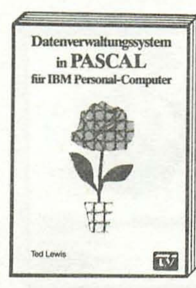
Software & Computer-Bücher

























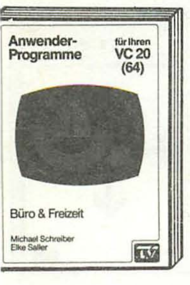






















Zu diesem Buch ist eine Diskette **zusätzlich** erhältlich

Dieses Buch beseitigt die Schwachstellen von Apple-PASCAL. Es wendet sich an die Programmierer, denen die vorhandenen Möglichkeiten nicht ausreichen. Der PASCAL-Könner erhält weit über die Handbücher hinausgehende Informationen, und dem Anfänger wird durch fertige Listings und Routinen die Arbeit erleichtert.
Die einzelnen Kapitel behandeln Utilities für das Betriebssystem, Spracherweiterungen, Grafik und Textverarbeitung. Es wird eingegangen auf den Aufbau des Directories und verschiedenen Filetypen, wobei auch ein neuer Filetyp, „FOTO", eingeführt wird. Weiterhin wird die Bearbeitung von DOS 3.3-Disketten unter PASCAL beschrieben. An Spracherweiterungen gibt es z.B. PEEK/POKE-Routinen oder Datumsveränderung. Schließlich ist eine komplette Liste des internen Systemroutinen und deren Adressen vorhanden. Im Grafikteil wird u.a. ein Grafikeditor vorgestellt und die Benutzung der Blockgrafik in PASCAL-Programmen ermöglicht. Der Textverarbeitungsteil enthält z.B. ein erweiterbares Formatierprogramm und ein Hilfsprogramm zur Veränderung der Editor-Marken.
In jedem der fast 30 Kapitel wird beschrieben, wie der jeweilige Trick realisiert wird, worauf sich ein abtippfertiges Listing anschließt.

ISBN 3–620–00103–0

